The Dissertation Committee for Matthew Troy Haley certifies that this is the approved version of the following dissertation:

Traveling Waves and Impact Parameter Correlations in QCD Beyond the 1D Approximation

Committee:

Charles Chiu, Supervisor

Duane Dicus

Oscar Gonzales

Christina Markert

E. C. George Sudarshan

Traveling Waves and Impact Parameter Correlations in QCD Beyond the 1D Approximation

by

Matthew Troy Haley, B.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August, 2011

Dedicated to my parents, who planted the germ of everything I have accomplished.

Acknowledgments

I would like to thank my adviser Dr. Charles Chiu for his tireless efforts in helping me to reach this goal. I would also like to thank my colleague Man Fung Cheung for his valuable input. Special thanks are due to my wife, Sarah Haley, as well for her ever-present support of my endeavors over the years.

Traveling Waves and Impact Parameter Correlations in QCD Beyond the 1D Approximation

Matthew Troy Haley, Ph.D
The University of Texas at Austin, 2011

Supervisor: Charles B. Chiu

The theory of quantum chromodynamics (QCD) predicts that at high energies, such as those investigated in deep inelastic scattering experiments, hadrons evolve into dense gluonic states described by the BFKL equation, and at very high densities, the more general BK equation. In certain approximations, the BK equation reduces to a well studied reaction-diffusion type nonlinear partial differential equation, the FKPP equation, for which analytical results are known. In this work, we model the BK equation using a classical branching process rooted in the dipole model of QCD evolution. Because the BK equation is inherently two dimensional, our model allows dipole impact parameters to occupy the full transverse space. A one dimensional limit of this model is studied as well. Results are compared with the predictions of the FKPP equation, and correlations between evolution at different impact parameters are presented. The general features of previously studied one dimensional impact parameter models are verified, but the details are refined in what we believe to be a more accurate model.

Contents

List of Figures

Part I

Background

1 Introduction

Much effort has been applied to the understanding of a hadron's transition from a dilute parton gas to a saturated CGC (Color Glass Condensate). While the DGLAP (Dokshitzer, Gribov, Lipatov, Altarelli, Parisi) equation could explain data collected at DESY-HERA at very large Q^2, the investigation of the scaling region at moderate Q^2 and very small $x \sim Q^2/(Q^2+s)$ prompted the application of the integro-differential BFKL (Balitsky, Fadin, Kuraev, Lipatov) equation, which resums infrared logarithms ($\log 1/x$). [**?**]

In its original formulation, the BFKL equation can be derived from the infinite sum of ladder diagrams of Reggeized gluons in the t-channel, as described in [6]. This derivation is known as the "BFKL pomeron" or "hard pomeron", giving the Regge trajectory $\alpha_P(t) = 1+4\bar{\alpha}_s \ln 2$. However, in the mid 90s, Mueller was able to rederive the BFKL equation in a much simpler s-channel picture and show that the BFKL pomeron is equivalent to a formulation describing dipole splittings in transverse space[11, 12]. A set of color dipoles comprise a so-called onium configuration, in which the emission of new gluons gives rise to new dipoles. Evolution consists of "parent" dipoles splitting into "daughter" dipoles with a characteristic probability

$$\frac{dP_{x_{01}\to x_{02},x_{12}}}{dY} = \frac{x_{01}^2 d^2\mathbf{x}_2}{x_{02}^2 x_{12}^2} \tag{1}$$

The amplitude of a photonic probe interacting with such a highly evolved hadron is roughly proportional to the number of dipoles in the hadron having the same approximate impact parameter and size as the $q\bar{q}$ dipole into which the probe splits.

In the context of the dipole model, the BK (Balitsky, Kovchegov) equation–essentially the BFKL equation modified by a nonlinear term responsible for saturation in the CGC regime–has been studied in a variety of analytical and computational ways in the past decade. The full BK equation in transverse space reads

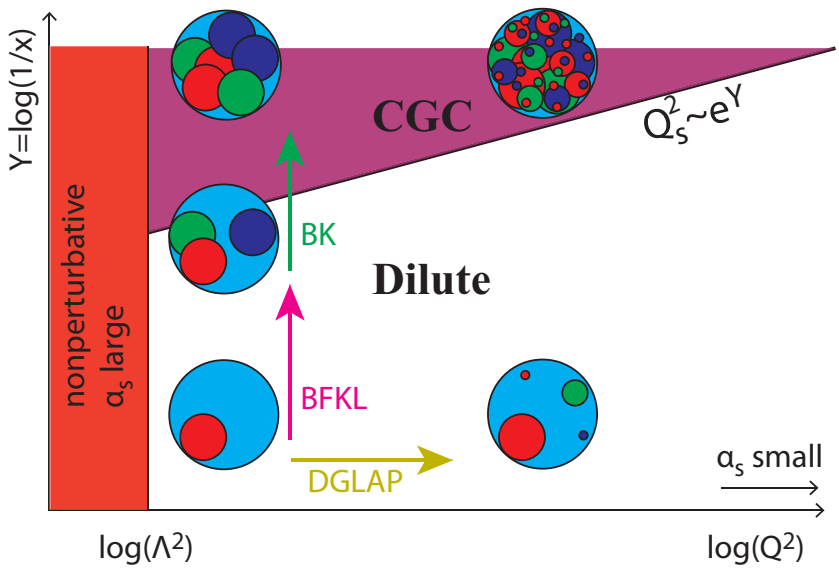

Figure 1: Phase diagram of a hadron in deep inelastic scattering

$$\frac{\partial N(\mathbf{x}_{01}, Y)}{\partial Y} = \frac{\bar{\alpha}}{2\pi} \int_\rho d^2\mathbf{x}_2 \frac{x_{01}^2}{x_{02}^2 x_{12}^2} 2N(\mathbf{x}_{02}, Y) - N(\mathbf{x}_{01}, Y) - N(\mathbf{x}_{02}, Y)N(\mathbf{x}_{12}, Y) \quad (2)$$

The first two terms on the right hand side represent the increase in the amplitude due to branching diffusion, the third term a virtual correction necessary to normalize the onium wavefunction [17], and the final term the nonlinearity that restores unitarity to the BFKL equation.

One of the most exciting theoretical developments of the past decade has been the discovery that for fixed impact parameter collisions, the BK equation belongs to the universality class of the FKPP (Fisher Kolmogorov, Petrovsky, Piscounov) equation[14, 15]. That is, an analogy was noted between high energy QCD evolution and a well studied reaction-diffusion equation. With the appropriate transformations, the scattering amplitude can be put into the form

$$\partial_t u(t,x) = \partial_x^2 u(t,x) + u(t,x) - u^2(t,x) \quad (3)$$

the solution of which describes a traveling wave. The time, t, is analogous to the rapidity, Y, and spatial coordinate x to the dipole momentum. It is thereby possible to speak of a saturation wave front, ρ_s, that travels to smaller dipole sizes as collision

energy increases.

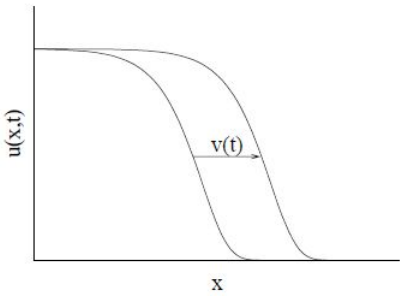

Figure 2: Traveling wave solution to the FKPP equation [2]

An important caveat to the application of the FKPP equation is that it is a mean field limit of the true stochastic evolution equations. Due to the discrete nature of an onium state consisting of a finite number of dipoles, fluctuations in dipole number must play a role in the evolution. Because the true stochastic equations are not known and their formulation would probably require a more sophisticated understanding of the saturation mechanism than is presently available, many researchers have taken to monte carlo computer modeling of stochastic splittings. This continues to be a very active field of research [20, 21, 22, 23, 24, 25]. Two of the most recent of these in particular [24, 25] have informed the study described in this manuscript. It will be explained what has been accomplished so far and how it can be extended using a full two dimensional model.

2 History of the BFKL Equation

2.1 Regge Theory and the origins of the Pomeron

Before the advent of QCD, a variety of other approaches were used to study strong interactions, some of which are still useful today. Regge theory, a branch of S-matrix theory, was for instance successfully used to predict the rise of hadronic cross-sections at small x, or increasing center of mass energy. During the sixties when the fundamentals of strong interactions were not yet known, studies focused on the exchange of massive mesons, as in the Yukawa theory of nuclear force. At that time it was postulated (by Chew and Frautschi [3, 4], for example) that there were no elementary strongly interacting particles among hadrons, i.e. mesons and baryons, as it appeared as a consequence of Regge Theory that all hadrons are bound states or resonances with interlocking angular momentum states. To this end a substantial attempt was made to explain all of strong interactions through studying the implications of a number of assumptions about the S-matrix. The argument was that if the strongly interacting particles that were known obeyed a self-consistent theory of the S-matrix, then the need for elementary particles of the strong force would be obviated, yielding a "bootstrap" theory, as it was called.

It was not until detailed data of the nucleon structure functions was obtained from inelastic electron-proton scattering at Stanford Linear Accelerator in 1969 that the physics community came to accept the existence of spin $\frac{1}{2}$ "partons", as Feynman dubbed them, which comprise the nucleon. Although this marked the shift toward what was the beginning of QCD (and the decreasing popularity of the S-matrix approach, especially with regards to phenomenology), it is worth reflecting of the substantial successes of S-matrix theory and how they have shed light on much later developments in QCD. Some insights from S-matrix theory still await a proper QCD treatment while others lie beyond the reach of a perturbative theory like QCD.

We will now give an abbreviated tour of Regge theory, in which amplitudes of strong interaction processes are expanded in terms of partial waves:

$$A_{a\bar{c}\to\bar{b}d}(s,t) = \sum_{l=0}^{\infty}(2l+1)a_l(s)P_l(1+2t/s) \tag{4}$$

or by crossing symmetry,

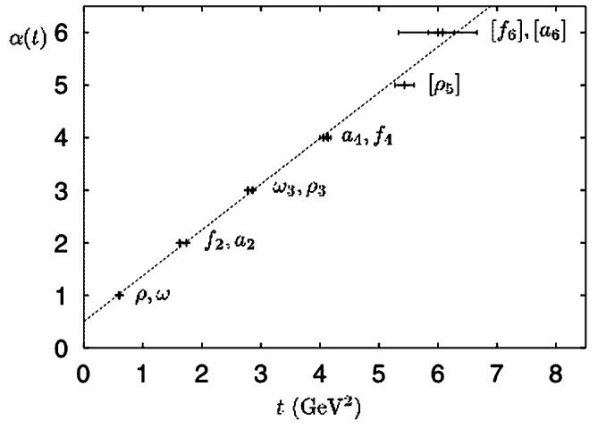

Figure 3: The Chew Frautschi plot of mesons' mass squared versus spin. [7]

$$A_{ab\to cd}(s,t) = \sum_{l=0}^{\infty}(2l+1)a_l(t)P_l(1+2s/t) \tag{5}$$

where $P_l(z)$ are Legendre polynomials and $a_l(s)$ are called a partial wave amplitudes. (5) can be rewritten as a contour integral in the complex angular momentum plane in what is known as a Sommerfeld-Watson transform. The contour surrounds the positive x-axis so that the residues reproduce the sum in (5):

$$A(s,t) = \frac{1}{2i}\oint_C dl(2l+1)\frac{a(l,t)}{\sin\pi l}P(l,1+2s/t) \tag{6}$$

$a(l,t)$ and $P(l,1+2s/t)$ are analytic continuations of the functions in (5). If we consider the Regge region $s \gg |t|$, we can expand $P_l(z)$ as

$$P_l(1+2s/t) \overset{s\gg t}{\to} \frac{\Gamma(2l+1)}{\Gamma^2(l+1)}\left(\frac{s}{2t}\right)^l \tag{7}$$

This allows us to conveniently deform the contour in (6) to a vertical line on which $\Re\mathrm{e}(l) < 0$, causing $\left(\frac{s}{2t}\right)^l$ to vanish at large s. In the process of deforming the contour, however, we pick up poles in the l plane known as Regge poles. The residue of the pole with the largest real part leads to the amplitude behavior

$$A(s,t) \overset{s\to\infty}{\sim} s^{\alpha(t)} \tag{8}$$

Recalling that $\alpha(t)$ is an angular momentum, one can learn about this function by

plotting low lying mesons with spin J_i and mass m_i, as done on figure 3. It then becomes immediately obvious that $J_i = \alpha(m_i^2)$ is a linear function, i.e. $\alpha(t) = \alpha(0) + \alpha' t$. The intercept of this plot has a special meaning: the optical theorem at large s gives the forward total cross-section as

$$\sigma_{tot} \propto s^{\alpha(0)-1} \tag{9}$$

Thus, the Regge intercept determines the total cross section. From figure 3, it appears the intercept is about .5, implying that the so-called Reggeons in the figure contribute

$$\sigma_{tot} \propto s^{-0.5} \tag{10}$$

to the total cross-section. But this is not at all what is observed! Instead, data shows that cross-sections rise starting at $\sqrt{s} \gtrsim 10$ GeV. In the late 1950s, Pomeranchuk proved that any scattering process in which there is charge exchange exhibits an asymptotically vanishing cross-section. Therefore, there must be a exchange with vacuum quantum numbers that causes the cross-section to rise. This Regge trajectory is called the Pomeron[1]. Later after the advent of QCD, it was conjectured that the integer values of the Pomeron trajectory $\alpha_{\mathbb{P}}(t)$ might correspond to bound states of gluons, or glueballs. Proving the existence of such entities remains one of the great remaining experimental challenges of high energy QCD.

2.2 The hard Pomeron attained through QCD ladder diagrams

Once perturbative QCD techniques had become well established, it was naturally wondered whether Pomeron behavior could be derived from pQCD. Copious detail on this program can be found in [6], the results of which we will now briefly touch on. Computing infinite ladder diagrams such as figure 4 left can reproduce the Pomeron behavior of (9). Slashes through vertical gluons indicate they have been "Reggeized", i.e. each is a sum of infinite ladder rungs such that the gluon propagator is replaced by

[1]Fits to the data actually indicate the presence of two kinds of Pomeron: a "soft" Pomeron with behavior $s^{0.08}$ and a "hard" pomeron with behavior $s^{0.4}$. Because the soft Pomeron lies outside the reach of perturbative methods, we will only focus on the hard Pomeron. [5]

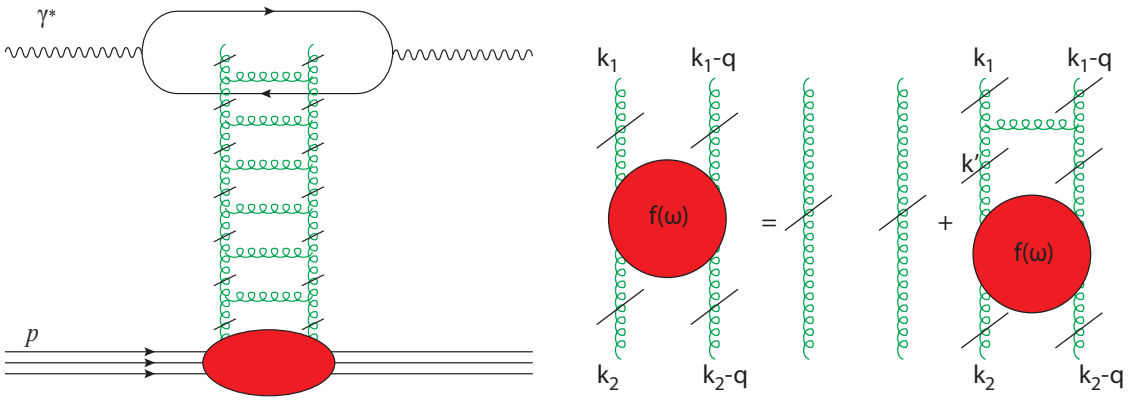

Figure 4: Left: A ladder diagram of Reggeized gluons representing Pomeron exchange. Right: Diagram illustrating the integral equation for the Mellin transformed amplitude, $f(\omega)$.

$$\tilde{D}_{\mu\nu}(s_i, k_i^2) = \frac{i g_{\mu\nu}}{\mathbf{k}_i^2} \left(\frac{s_i}{\mathbf{k}^2}\right)^{\epsilon(k_i^2)} \tag{11}$$

where i stands for the ith rung and $s_i = (k_{i-1} - k_{i+1})^2$ is the squared center of mass energy coming into the ith rung.

One may write an integral equation shown diagrammatically in figure 4 right and solve for the Mellin transformed amplitude at zero momentum transfer, $f(\omega, \mathbf{k}_1, \mathbf{k}_2, \mathbf{0})$, as such: [6]

$$f(\omega, \mathbf{k}_1, \mathbf{k}_2, \mathbf{0}) \approx \frac{1}{\pi k_1 k_2} \int_{-\infty}^{\infty} \frac{d\nu}{2\pi} \left(\frac{k_1^2}{k_2^2}\right)^{i\nu} \frac{1}{\omega - \omega_0 + a^2\nu^2} \tag{12}$$

with

$$\omega_0 = 4\bar{\alpha}_s \ln 2 \tag{13}$$

and ν the anomalous dimension of the BFKL eigenvalue function, which we will later cover in detail. Performing the contour integration and inverting the Mellin transform,

$$F(s, \mathbf{k}_1, \mathbf{k}_2, \mathbf{0}) \approx \frac{1}{\sqrt{\mathbf{k}_1^2 \mathbf{k}_2^2}} \left(\frac{s}{k^2}\right)^{\omega_0} \frac{1}{\sqrt{\pi \ln(s/k^2)}} \frac{1}{2\pi a} \exp\left(-\frac{\ln^2(\mathbf{k}_1^2/\mathbf{k}_2^2)}{4a^2 \ln(s/k^2)}\right) \tag{14}$$

The full $q\bar{q}$ forward elastic scattering amplitude is then

$$\frac{\mathcal{A}^{(1)}(s,0)}{s} = 4i\alpha_s^2 \delta_{\lambda_1'\lambda_1}\delta_{\lambda_2'\lambda_2} G_0^{(1)} \int \frac{d^2\mathbf{k}_1}{k_1^2}\frac{d^2\mathbf{k}_2}{k_2^2} F(s,\mathbf{k}_1,\mathbf{k}_2,\mathbf{0}) \tag{15}$$

and thus,

$$\sigma_{tot} \sim s^{\omega_0} = s^{\alpha_{\mathbb{P}}(0)-1} = s^{4\bar{\alpha}_s \ln 2} \tag{16}$$

So we see that the pQCD ladder diagram calculation successfully predicts the Pomeron trajectory required for the rise of the total cross-section.

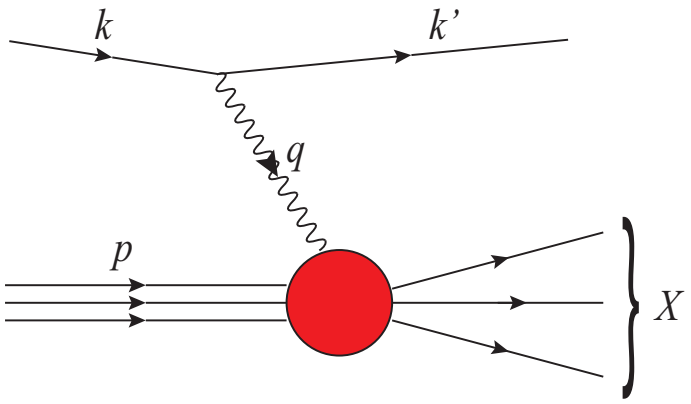

Figure 5: Inclusive deep inelastic scattering for $e^- p \to e^- X$

3 Dipole Formulation of BFKL Equation

3.1 Description of the dipole model

So far we have looked at the BFKL equation from the standpoint of t-channel interactions of $\gamma^* p \to X$. However, a much simpler method of deriving the BFKL equation was achieved in the s-channel picture by Mueller [11], in which the evolution takes place in the target as one boosts it to greater rapidity. In this approach, the target interacts with the probe as an "onium" state of quantum fluctuations. An onium comprises a high occupancy Fock state when the interaction energy is large. Using the onium wavefunction to calculate the dipole cross-section, other useful deep inelastic scattering observables may be calculated.

The idea for calculating the dipole cross-section had been popular before Mueller used it to rederive the BFKL equation [33][34]. In a process such as $e^- p \to e^- X$ (see figure 5), the dominant contribution to the scattering cross-section comes from photon's dissociation into a quark-antiquark color-singlet state that strongly interacts with the proton (see figure 6). This approach is only legitimate when the dissociation time of the photon is large compared to interaction time with the proton. We can estimate these times using energy uncertainty as follows [7]. Let the four-momentum of the photon, quark, and antiquark be, respectively,

$$q = (q_0,0,0,q_3) \quad k_1 = (E_1, \mathbf{k}_T, zq_3) \quad k_2 = (E_2, -\mathbf{k}_T, (1-z)q_3 \tag{17}$$

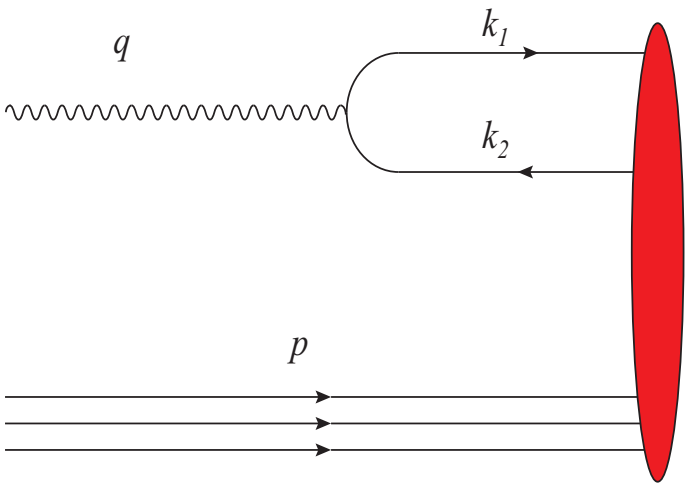

Figure 6: Photon dissociation into quark-antiquark pair and interaction with hadron. A cut of the total cross-section is displayed.

where z is the fraction of the photon momentum carried by the quark ($0 \leq z \leq 1$), and $\mathbf{k}_T$ is the two dimensional transverse momentum of the quark. The dissociation time for the photon is then given by

$$\tau_{dis} = \frac{1}{|q_0 - E_1 - E_2|} \tag{18}$$

Using expansions in the large q_3 limit, $E_1 \approx zq_3 + \frac{m_f^2 + \mathbf{k}_T^2}{2zq_3}$, $E_2 \approx (1-z)q_3 + \frac{m_f^2 + \mathbf{k}_T^2}{2(1-z)q_3}$, $q_0 \approx q_3 - \frac{Q^2}{2q_3}$,

$$\tau_{dis} \approx \frac{1}{\left| -\frac{Q^2}{2q_0} - \frac{m_f^2 + \mathbf{k}_T^2}{2z(1-z)q_0} \right|} \tag{19}$$

If we take the interaction time of the dissociated photon with the proton in its rest frame to be of the order of the proton confinement radius $1/\Lambda$, and set $|\mathbf{k}_T| \approx \Lambda$, our timescale comparison yields

$$\tau_{dis} \gg \frac{1}{\Lambda}$$

$$2q_0 m_p \gg \frac{m_p}{\Lambda}\left(Q^2 + \frac{m_f^2 + \Lambda^2}{z(1-z)}\right)$$

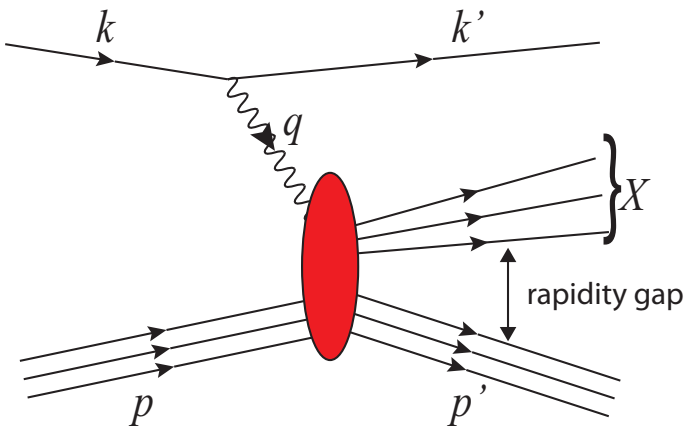

Figure 7: Diffractive deep inelastic scattering

$$W^2 \gg \frac{m_p}{\Lambda}\left(Q^2 + \frac{m_f^2 + \Lambda^2}{z(1-z)}\right) \tag{20}$$

since $W^2 = (p+q)^2 = m_p^2 - Q^2 + 2m_p q_0$ in the proton rest frame. (20) tells us that unless z is close to 0 or 1, $W^2/Q^2 \gg 1$. This condition has a special significance in deep inelastic scattering–recalling the definition of the Bjorken x,

$$x := \frac{Q^2}{2p\cdot q} = \frac{Q^2}{(p+q)^2 - m_p^2 - q^2} \approx \frac{Q^2}{W^2 + Q^2} \tag{21}$$

We see that $W^2/Q^2 \gg 1$ at large energies implies we are in the small x regime. Therefore, for the high energy processes we will be considering, the dipole picture is appropriate. Note that this method differs from the usual deep inelastic picture in which a parton is knocked out by the virtual photon in that the dipole is interacting with the gluonic field of the hadron, as opposed to a single parton.

Deep inelastic scattering experiments, such as HERA, have been among the most fruitful for the application of the dipole model. Deep inelastic scattering itself is good testing ground for high energy QCD since the photon kinematics are contained in the measurement of the outgoing lepton, yielding Q^2. Models for the dipole cross-section have successfully been applied to inclusive and diffractive events at HERA [35, 36, 37] (see figure 7 for an illustration of the latter).

In order to derive QCD evolution equations, we should focus our attention on the wavefunction of the onium state of the target hadron. This state is built from succes-

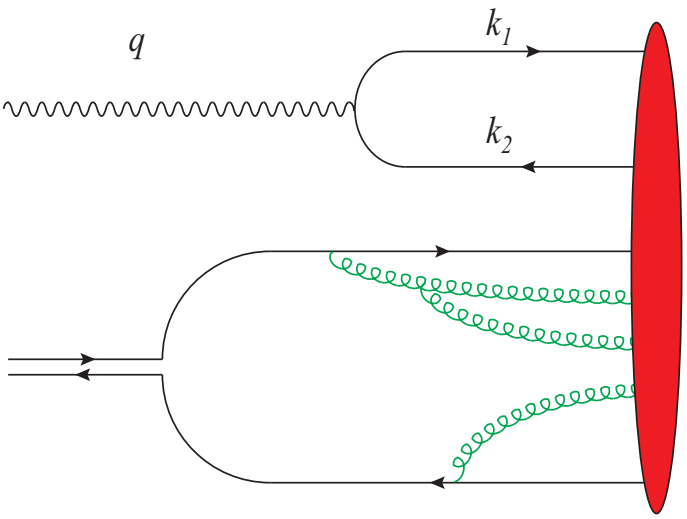

Figure 8: Quark-antiquark pair interacting with an evolved target. A cut of the total cross-section is displayed.

sive splittings of the original valence partons of the target until dense gluonic states comprise the target at high energy. This process is called a gluonic cascade, a still shot of which is shown in figure 8. Because quarks or gluons splitting into a gluon exhibit a logarithmic singularity in z [1], soft gluons dominate in the small x limit or alternatively in the large rapidity limit, as $y = \ln 1/x$. In the limit of large number of colors (N_c), each emitted gluon is treated as a zero-size quark-antiquark pair[2], as shown in figure 9. Note, however, that the dipoles are of *finite* size, as can also be seen in the figure. This is a potential source of confusion, as we usually think of a dipole as being the limit of zero separation between a charge and anti-charge, although in this case the color dipoles are finite size.

A major advantage to the dipole-onium interaction model is that the cross-section for the subprocess shown in figure 8 factorizes:

$$\sigma^{\gamma^* p}(Y, Q^2) = \int d^2b d^2x_{01} \int_0^1 dz \, |\psi_{\gamma^*}(z, x_{01}Q)|^2 \, \sigma_{dipole}(Y, x_{01}) \tag{22}$$

where $\psi_{\gamma^*}(z, x_{01}Q)$ is the photon wavefunction for splitting into a quark-antiquark dipole of size x_{01}, z the longitudinal momentum fraction of the quark, and σ_{dipole} the dipole forward scattering amplitude.

[2]This is related to T'Hooft's observation that for SU(N), as N$\to \infty$ planar graphs dominate over those of differing topology [49].

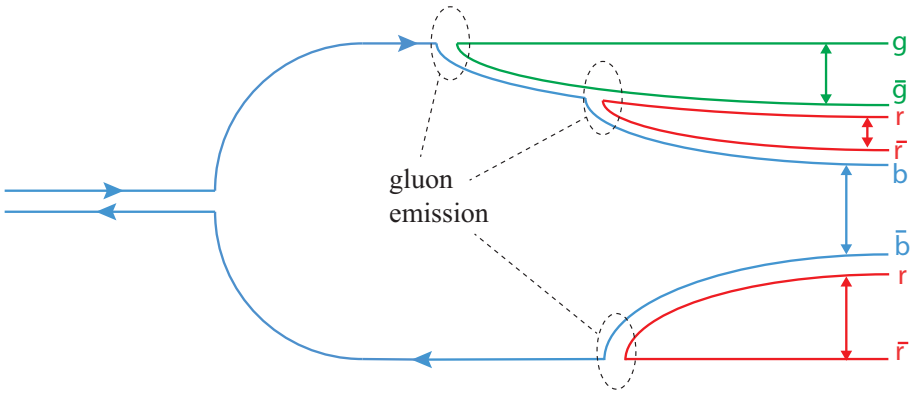

Figure 9: Illustration of dipoles in the evolved target from figure 8. Each dipole is indicated by a double-headed arrow.

3.2 QCD evolution using color dipoles

3.2.1 Single emitted gluon wavefunction

With the dipole model of hadron evolution we can now see how QCD evolution equations, in particular the BFKL equation, can be obtained. We will follow the seminal paper by Mueller [11] with the addition of some omitted details. The accuracy of our calculation will be leading logarithmic such that the $\left(\alpha \ln \frac{1}{z_0}\right)^n$ contribution to the square of the onium wavefunction will be computed for n soft gluons with momentum between $z_0 p$ and p. Using the usual Feynman rules for a gluon and quark vertex, the diagrams in figure 10 yield the following contribution to the momentum space onium wavefunction:

$$\psi^{(1)a}_{\alpha\beta}(\mathbf{k}_1, \mathbf{k}_2; z_1, z_2) = -gT^a \left[\psi^{(0)}_{\alpha\beta}(\mathbf{k}_1; z_1) - \psi^{(0)}_{\alpha\beta}(\mathbf{k}_1 + \mathbf{k}_2; z_1)\right] \frac{\mathbf{k}_2 \cdot \epsilon_2^\lambda}{k_2^2} \tag{23}$$

where a is the color index of the emitted gluon, T^a the SU(3) generator, α and β spinor indices, $z_n := k_n^+/p^+$ the fractional momentum of the original quark-antiquark pair (in lightcone coordinates), ϵ_2^λ the polarization vector of the emitted gluon with helicity λ, and $\psi^{(n)}$ is the wavefunction when n soft gluons have been emitted.

We will now transform the momentum space wavefunction to transverse space where a significant simplification takes place: in the high energy limit the emission of small z, or soft, gluons dominates, and the transverse coordinates of the parent partons are not affected by subsequent evolution of the system. Thus, each dipole evolves

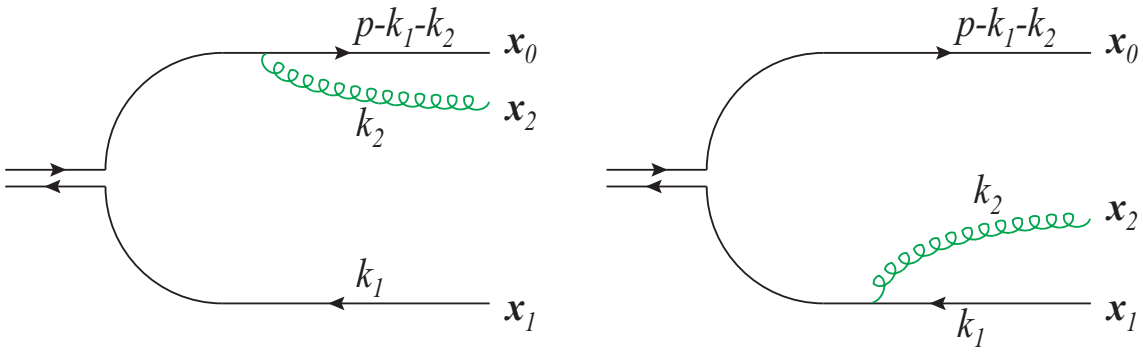

Figure 10: Single gluon emission from quark-antiquark pair

independently of the others. Their transverse coordinates are said to be "frozen". Fourier transforming to transverse space,

$$\psi^{(1)a}_{\alpha\beta}(\mathbf{x}_1,\mathbf{x}_2;z_1,z_2)=\int\frac{d^2\mathbf{k}_2}{(2\pi)^2}\int\frac{d^2\mathbf{k}_1}{(2\pi)^2}e^{i\mathbf{k}_1\cdot\mathbf{x}_1+i\mathbf{k}_2\cdot\mathbf{x}_2}\psi^{(1)a}_{\alpha\beta}(\mathbf{k}_1,\mathbf{k}_2;z_1,z_2) \tag{24}$$

Substituting (23) into (24),

$$=gT^a\int\frac{d^2\mathbf{k}_2}{(2\pi)^2}e^{i\mathbf{k}_2\cdot\mathbf{x}_2}\left(\psi^{(0)}_{\alpha\beta}(\mathbf{x}_1;z_1)-\int\frac{d^2\mathbf{k}_1^{'}}{(2\pi)^2}e^{i(\mathbf{k}_1^{'}-\mathbf{k}_2)\cdot\mathbf{x}_1}\psi^{(0)}_{\alpha\beta}(\mathbf{k}_1^{'};z_1)\right)\frac{\mathbf{k}_2\cdot\epsilon_2^{\lambda}}{k_2^2}$$

$$=gT^a\psi^{(0)}_{\alpha\beta}(\mathbf{x}_1;z_1)\int\frac{d^2\mathbf{k}_2}{(2\pi)^2}\left(e^{i\mathbf{k}_2\cdot(\mathbf{x}_2-\mathbf{x}_0)}-e^{i\mathbf{k}_2\cdot(\mathbf{x}_2-\mathbf{x}_1)}\right)\frac{\mathbf{k}_2\cdot\epsilon_2^{\lambda}}{k_2^2} \tag{25}$$

At this point we will need to prove the following Hankel transform:

$$\int d^2\mathbf{k}e^{i\mathbf{k}\cdot\mathbf{x}}\frac{\mathbf{k}\cdot\epsilon}{k^2}=-2\pi i\frac{\mathbf{x}\cdot\epsilon}{x^2} \tag{26}$$

We can demonstrate (26) as follows:

$$
\begin{aligned}
LHS &= \sum_{j=1,2} \int d^2\mathbf{k} e^{i\mathbf{k}\cdot\mathbf{x}} \frac{k_j \epsilon_j}{k^2} \\
&= -i \sum_{j=1,2} \frac{\partial}{\partial x_j} \int d^2\mathbf{k} e^{i\mathbf{k}\cdot\mathbf{x}} \frac{\epsilon_j}{k^2} \\
&= 2\pi i \sum_{j=1,2} \hat{e}_j \cdot \nabla_x \int_0^\infty dk J_0(kx) \frac{\epsilon_j}{k} \\
&= -2\pi i \sum_{j=1,2} \hat{e}_j \cdot \hat{x} \int_0^\infty dk J_1(kx) \epsilon_j \\
&= -2\pi i \sum_{j=1,2} \frac{\hat{e}_j \cdot \hat{x} \epsilon_j}{x} \\
&= -2\pi i \frac{\mathbf{x}\cdot\epsilon}{x^2} \qquad \square
\end{aligned}
$$

Using (26) in (25), we obtain

$$
= -\frac{igT^a}{2\pi} \psi^{(0)}_{\alpha\beta}(\mathbf{x}_1; z_1) \left(\frac{\mathbf{x}_{20}}{x_{20}^2} - \frac{\mathbf{x}_{21}}{x_{21}^2} \right) \cdot \epsilon_2^\lambda \tag{27}
$$

where a Hankel transform has been performed in the last step. Note that $\mathbf{x}_0 = 0$ in the above, and $\mathbf{x}_{20} := \mathbf{x}_2 - \mathbf{x}_0$, $\mathbf{x}_{21} := \mathbf{x}_2 - \mathbf{x}_1$. Now let us calculate the squared and summed wavefunction. If the squared and summed wavefunction for zero gluons present is

$$
\Phi^{(0)}(\mathbf{x}_1, z_1) := \sum_{\alpha\beta} \left| \psi^{(0)}_{\alpha\beta}(\mathbf{x}, z_1) \right|^2 \tag{28}
$$

then similarly, that for one gluon present is

$$
\Phi^{(1)}(\mathbf{x}_1, z_1) := \int d^2\mathbf{x}_2 \int_{z_0}^{z_1} \frac{dz_2}{z_2} \sum_{\alpha\beta} \frac{1}{2} \sum_{\lambda=1,2} \sum_a \left| \psi^{(1)a}_{\alpha\beta}(\mathbf{x}_1, \mathbf{x}_2; z_1, z_2) \right|^2
$$

z_0 serves as a lower cutoff to the emitted gluon momentum, z_2. The largest momentum the gluon can possess is z_1 in the leading logarithmic approximation.

$$= \frac{1}{2}\frac{g^2}{(2\pi)^2}\sum_a T^a T^a \int d^2\mathbf{x}_2 \int_{z_0}^{z_1}\frac{dz_2}{z_2}\sum_{\alpha\beta}\left|\psi^{(0)}_{\alpha\beta}(\mathbf{x},z_1)\right|^2 \sum_{\lambda=1,2}\left[\left(\frac{\mathbf{x}_{20}}{x_{20}^2}-\frac{\mathbf{x}_{21}}{x_{21}^2}\right)\cdot\epsilon_2^\lambda\right]^2$$

$$= \frac{\alpha N_c}{\pi}\int\frac{d^2\mathbf{x}_2}{2\pi}\int_{z_0}^{z_1}\frac{dz_2}{z_2}\Phi^{(0)}(\mathbf{x}_1,z_1)\left(\frac{\mathbf{x}_{20}}{x_{20}^2}-\frac{\mathbf{x}_{21}}{x_{21}^2}\right)^2 \tag{29}$$

where we have used the strong coupling constant $\alpha_s = \frac{g^2}{4}$, the trace over $\sum_a T^a T^a = N$ in the adjoint representation of SU(N), and the polarization sum was evaluated with $\epsilon^1 = (0,1,0,0)$ and $\epsilon^2 = (0,0,1,0)$. After foiling the term in parenthesis in (29) and some algebraic simplification we arrive at

$$\Phi^{(1)}(\mathbf{x}_1,z_1) = \frac{\alpha N_c}{\pi}\int\frac{d^2\mathbf{x}_2}{(2\pi)}\int_{z_0}^{z_1}\frac{dz_2}{z_2}\frac{x_{10}^2}{x_{20}^2x_{21}^2}\Phi^{(0)}(\mathbf{x}_1,z_1) \tag{30}$$

At this point we might want to pause to see what we have gained. Notice that the momentum space representation of single gluon emission,

$$\Phi^{(1)}(\mathbf{k}_1,z_1) = \frac{1}{(2\pi)^2}\int d^2\mathbf{k}_2\int_{z_0}^{z_1}\frac{dz_2}{z_2}\frac{1}{2}\sum_{\lambda,a,\alpha\beta}\left|\psi^{(1)a}_{\alpha\beta}(\mathbf{k}_1,\mathbf{k}_2,z_1,z_2)\right|^2 \tag{31}$$

does not exhibit the same clean factorization as (30), which is written as an integral of the zero gluon, bare quark-antiquark wavefunction squared. The simplicity of (30) will allow us to generalize the onium wavefunction to include n soft gluons. Also, we will see the kernel of the spatial integral, $x_{10}^2/x_{20}^2x_{21}^2$ play a significant role later in this manuscript.

3.2.2 n emitted gluon wavefunction

For notational simplicity, let us make use of the following Jacobian,

$$d^2\mathbf{x}_2 = x_2 dx_2 d\phi = J dx_{12} dx_{20} \tag{32}$$

where ϕ is the angle between $\mathbf{x}_{20}$ and $\mathbf{x}_{10}$. Inserting an extra factor of 2 to account for the $0<\phi<\pi$ as well as the $\pi<\phi<2\pi$ domain,

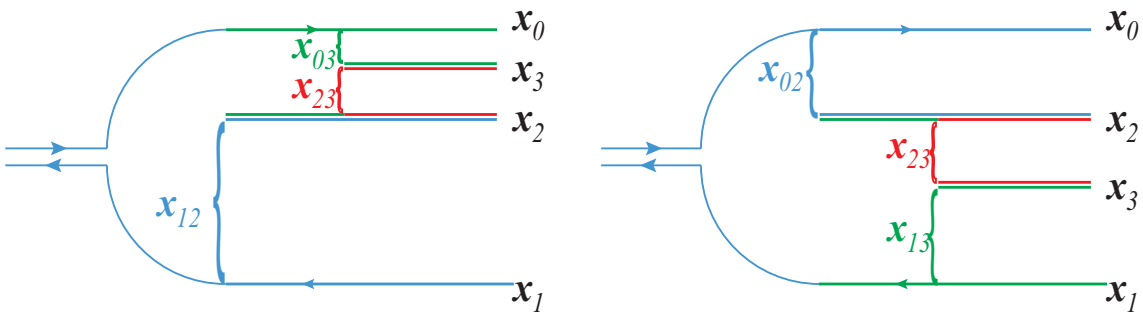

Figure 11: Diagrams for two gluon emission

$$J(x_{12}, x_{02}) = \frac{4x_{21}x_{20}}{\sqrt{[(x_{21}+x_{20})^2 - x_{10}^2]\,[x_{10}^2 - (x_{21}-x_{20})^2]}} \tag{33}$$

For the 2 gluon emitted squared wavefunction, the second gluon can be emitted from either the x_{02} dipole (lefthand picture in figure 11) or the x_{12} dipole (righthand picture in figure 11). Given these two possibilities, the 2 gluon squared wavefunction can then be written,

$$\begin{aligned}
\Phi^{(2)}(\mathbf{x}_1, z_1) &= \left(\frac{\alpha N_c}{2\pi^2}\right)^2 \int d^2\mathbf{x}_2 \int_{z_0}^{z_1} \frac{dz_2}{z_2} \frac{x_{10}^2}{x_{20}^2 x_{21}^2} \int d^2\mathbf{x}_3 \int_{z_0}^{z_1} \frac{dz_3}{z_3} \left(\frac{x_{02}^2}{x_{30}^2 x_{32}^2} + \frac{x_{12}^2}{x_{31}^2 x_{32}^2}\right) \Phi^{(0)}(\mathbf{x}_1, z_1) \\
&= \left(\frac{\alpha N_c}{2\pi^2}\right)^2 \ln^2\left(\frac{z_1}{z_0}\right) \Phi^{(0)}(\mathbf{x}_1, z_1) \int d^2\mathbf{x}_2 \frac{x_{10}^2}{x_{20}^2 x_{21}^2} \int d^2\mathbf{x}_3 \left(\frac{x_{02}^2}{x_{30}^2 x_{32}^2} + \frac{x_{12}^2}{x_{31}^2 x_{32}^2}\right)
\end{aligned} \tag{34}$$

Performing the transform of coordinates (32) using the Jacobian (33), we can also write this solution as

$$\begin{aligned}
&= \left(\frac{2\alpha N_c}{\pi^2}\right)^2 \ln^2\left(\frac{z_1}{z_0}\right) \Phi^{(0)}(\mathbf{x}_1, z_1) x_{10}^2 \int dx_{20} dx_{21} \frac{J(x_{20}, x_{21})}{x_{20}^2 x_{21}^2} \\
&\quad \times \left[\int dx_{30} dx_{32} \frac{J(x_{30}, x_{32}) x_{02}^2}{x_{30}^2 x_{32}^2} + \int dx_{32} dx_{31} \frac{J(x_{31}, x_{32}) x_{12}^2}{x_{31}^2 x_{32}^2}\right]
\end{aligned} \tag{35}$$

Now that we have calculated the squared wavefunctions for 1 and 2 soft gluons, we are prepared to generalize to n gluons through the use of a generating functional. Let $\Phi(\mathbf{x}_1, z_1, u(\mathbf{x}, z))$ be defined by the equation,

$$\frac{\delta}{\delta u(\mathbf{x}_2, z_2)} \frac{\delta}{\delta u(\mathbf{x}_3, z_3)} \cdots \frac{\delta}{\delta u(\mathbf{x}_{n+1}, z_{n+1})} \Phi(\mathbf{x}_1, z_1, u(\mathbf{x}, z))|_{u=0}$$
$$= \Phi^{(n)}(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{n+1}; z_2 \cdots z_{n+1}) \tag{36}$$

where $\Phi^{(n)}$ is the n gluon squared wavefunction, and $\mathbf{x}_{n+1}$, z_{n+1} are the transverse position and momentum fraction, respectively, of the nth gluon. Let us now define the generating functional Z by

$$\Phi(\mathbf{x}_1, z_1, u) = \Phi^{(0)}(\mathbf{x}_1, z_1) Z(\mathbf{x}_1, \mathbf{x}_0, z_1, u) \tag{37}$$

such that the following holds:

$$Z(\mathbf{x}_1, \mathbf{x}_0, z_1, u) = 1 + \frac{\alpha N_c}{2\pi^2} x_{01}^2 \int d^2\mathbf{x}_2 \frac{x_{10}^2}{x_{20}^2 x_{21}^2} \int_{z_0}^{z_1} u(\mathbf{x}_2, z_2) Z(\mathbf{x}_2, \mathbf{x}_1, z_2, u) Z(\mathbf{x}_2, \mathbf{x}_0, z_2, u) \tag{38}$$

Using the standard rules for functional differentiation,

$$\frac{\delta}{\delta u(\mathbf{x})} u(\mathbf{y}) = \delta^{(2)}(\mathbf{x} - \mathbf{y}) \qquad \frac{\delta}{\delta u(\mathbf{x})} \int d^2\mathbf{y} u(\mathbf{y}) f(\mathbf{y}) = f(\mathbf{x}) \tag{39}$$

we can demonstrate (36) by reproducing the 2 gluon squared wavefunction (34). Let us calculate the LHS of (36) before setting $u = 0$.

$$\frac{\delta}{\delta u(\mathbf{x}_3, z_3)} \frac{\delta}{\delta u(\mathbf{x}_2, z_2)} \Phi^{(0)}(\mathbf{x}_1, z_1) Z(\mathbf{x}_1, \mathbf{x}_0, z_1, u)$$

$$
\begin{aligned}
&= \Phi^{(0)}(\mathbf{x}_1, z_1) \frac{\delta}{\delta u(\mathbf{x}_3, z_3)} \frac{\delta}{\delta u(\mathbf{x}_2, z_2)} \left[1 + \frac{\alpha N_c}{2\pi^2} \int d^2\mathbf{x}_\alpha \frac{x_{10}^2}{x_{\alpha 0}^2 x_{\alpha 1}^2} \right. \\
&\qquad \left. \times \int_{z_0}^{z_1} \frac{dz_\alpha}{z_\alpha} u(\mathbf{x}_\alpha, z_\alpha) Z(\mathbf{x}_\alpha, \mathbf{x}_1, z_\alpha, u) Z(\mathbf{x}_\alpha, \mathbf{x}_0, z_\alpha, u) \right] \quad (40) \\
&= \frac{\alpha N_c}{2\pi^2} \frac{1}{z_2} \frac{x_{10}^2}{x_{20}^2 x_{21}^2} \Phi^{(0)}(\mathbf{x}_1, z_1) \frac{\delta}{\delta u(\mathbf{x}_3, z_3)} \left[Z(\mathbf{x}_2, \mathbf{x}_1, z_2, u) Z(\mathbf{x}_2, \mathbf{x}_0, z_2, u) \right] \\
&= \frac{\alpha N_c}{2\pi^2} \frac{1}{z_2} \frac{x_{10}^2}{x_{20}^2 x_{21}^2} \Phi^{(0)}(\mathbf{x}_1, z_1) \left[\frac{\delta Z(\mathbf{x}_2, \mathbf{x}_1, z_2, u)}{\delta u(\mathbf{x}_3, z_3)} Z(\mathbf{x}_2, \mathbf{x}_0, z_2, u) \right. \\
&\qquad \left. + \frac{\delta Z(\mathbf{x}_2, \mathbf{x}_0, z_2, u)}{\delta u(\mathbf{x}_3, z_3)} Z(\mathbf{x}_2, \mathbf{x}_1, z_2, u) \right] \quad (41) \\
&= \left(\frac{\alpha N_c}{2\pi^2} \right)^2 \frac{1}{z_2} \frac{1}{z_3} \frac{x_{10}^2}{x_{20}^2 x_{21}^2} \Phi^{(0)}(\mathbf{x}_1, z_1) \left[\frac{x_{21}^2}{x_{23}^2 x_{31}^2} Z_{23;3} Z_{13;3} Z_{20;2} \right. \quad (42) \\
&\qquad \left. + \frac{x_{20}^2}{x_{23}^2 x_{30}^2} Z_{23;3} Z_{31;3} Z_{21;2} \right]
\end{aligned}
$$

where hopefully the abbreviated notation for $Z_{\alpha\beta;\gamma} := Z(\mathbf{x}_\alpha, \mathbf{x}_\beta, z_\gamma, u)$ is clear. Now letting $u = 0$ in (42) so that $Z_{\alpha\beta;\gamma} = 1$, and taking the appropriate integrals, we obtain (34).

While (38) yields the n gluon squared wavefunctions upon functional differentiation, it fails to address virtual corrections and does not satisfy

$$
\int d^2\mathbf{x}_1 \int_0^1 dz_1 \Phi(\mathbf{x}_1, z_1, u)|_{u=1} = 1 \quad (43)
$$

Cutting off the ultraviolet divergences caused by x_{20} or x_{21} going to zero, we introduce a size cutoff $\rho \ll R_{target}$ such that $x_{20}, x_{21} \geq \rho$. By enforcing (43) at each order in α one can obtain the generating functional with virtual corrections,

$Z(\mathbf{x}_1, \mathbf{x}_0, z_1, u)$

$$= \exp\left[-\frac{2\alpha N_c}{\pi}\ln\left(\frac{x_{10}}{\rho}\right)\ln\left(\frac{z_1}{z_0}\right)\right] + \frac{\alpha N_c}{2\pi^2}\int_{z_0}^{z_1}\frac{dz_2}{z_2}$$
$$\times\int_\rho \exp\left[-\frac{2\alpha N_c}{\pi}\ln\left(\frac{x_{10}}{\rho}\right)\ln\left(\frac{z_1}{z_2}\right)\right]\frac{d^2\mathbf{x}_2 x_{10}^2}{x_{20}^2 x_{21}^2}u(\mathbf{x}_2,z_2)Z_{2,1;2}Z_{2,0;2} \qquad (44)$$

This equation represents a classical branching process and is exact in the leading logarithmic approximation. Another form of this equation we will use, letting $Y := \ln\left(\frac{z_1}{z_0}\right)$, $y := \ln\left(\frac{z_2}{z_0}\right)$, and $\bar{\alpha} := \frac{\alpha N_c}{\pi}$, is

$Z(\mathbf{x}_1, \mathbf{x}_0, z_1, u)$

$$= \exp\left[-2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)Y\right] + \frac{\bar{\alpha}}{2\pi}\int_{z_0}^{z_1}\frac{dz_2}{z_2}$$
$$\times\int_\rho \exp\left[-2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)(Y-y)\right]\frac{d^2\mathbf{x}_2 x_{10}^2}{x_{20}^2 x_{21}^2}u(\mathbf{x}_2,z_2)Z_{2,1;2}Z_{2,0;2} \qquad (45)$$

3.2.3 BFKL from the n gluon onium wavefunction

The generating functional in (45) can now be rewritten as an amplitude. Adding the two equal terms at first order in $\bar{\alpha}$ yields a factor of two in second term of the RHS below:

$T(x_{10}, z_1; Q, z)$

$$= \bar{\alpha}v(Q,x_{10})\exp\left[-2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)Y\right] + 2\bar{\alpha}\int_z^{z_1}\frac{dz_2}{z_2}$$
$$\times\int_\rho \exp\left[-2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)(Y-y)\right]\tilde{K}(x_{10},x_{12})dx_{12}T(x_{12},z_2;Q,z) \qquad (46)$$

where

$$\tilde{K}(x_{10}, x_{12}) = \frac{1}{2\pi} \int_\rho \frac{x_{10}^2}{x_{20}^2 x_{21}^2} J(x_{21}, x_{20}) dx_{20} \tag{47}$$

Let us now write $T(Y, Qx_{10}) := T(x_{10}, z_1; Q, z)$ as the (inverse) Mellin transform[3] of $T_\omega(Qx_{10})$.

$$T(Y, Qx_{10}) = \int_{c-i\infty}^{c+i\infty} \frac{d\omega}{2\pi i} e^{\omega Y} T_\omega(Qx_{10}) \tag{48}$$

This contour integral is a vertical line in the complex plane drawn such that c is greater than the real part of any singularities of T_ω. Note that the first term on the RHS of (46) can be written as

$$\bar{\alpha} v(Q, x_{10}) \exp\left[-2\bar{\alpha} \ln\left(\frac{x_{10}}{\rho}\right) Y\right] = \int_{c-i\infty}^{c+i\infty} \frac{d\omega}{2\pi i} e^{\omega Y} \frac{\bar{\alpha} v(Q, x_{10})}{\omega + 2\bar{\alpha} \ln\left(\frac{x_{10}}{\rho}\right)} \tag{49}$$

since the pole of $\omega = -2\bar{\alpha} \ln\left(\frac{x_{10}}{\rho}\right)$ leads to the residue on the LHS of the equation. Evaluating the second term on the RHS of (46),

$$2\bar{\alpha} \int_z^{z_1} \frac{dz_2}{z_2} \int_\rho \exp\left[-2\bar{\alpha} \ln\left(\frac{x_{10}}{\rho}\right)(Y - y)\right] \tilde{K}(x_{10}, x_{12}) dx_{12} T(x_{12}, z_2; Q, z)$$

$$= 2\bar{\alpha} \int_0^Y dy \int_\rho \exp\left[-2\bar{\alpha} \ln\left(\frac{x_{10}}{\rho}\right)(Y - y)\right] \tilde{K}(x_{10}, x_{12}) dx_{12} \int_{c-i\infty}^{c+i\infty} \frac{d\omega}{2\pi i} e^{\omega(y-Y)} T_\omega(Qx_{12})$$

$$= 2\bar{\alpha} \int_{c-i\infty}^{c+i\infty} \frac{d\omega}{2\pi i} \int_\rho \tilde{K}(x_{10}, x_{12}) dx_{12} \frac{1}{\omega + 2\bar{\alpha} \ln\left(\frac{x_{10}}{\rho}\right)}$$

$$\times \left\{1 - \exp\left[-\left(2\bar{\alpha} \ln\left(\frac{x_{10}}{\rho}\right) + \omega\right) Y\right]\right\} T_\omega(Qx_{12})$$

[3]Technically, this would be the inverse Laplace transform of $T_\omega(Qx_{10})$, but these transforms are related since $\left\{\mathcal{M}^{-1} T_\omega\right\}(e^{-Y}) = \frac{1}{2\pi i} \int_c e^{\omega Y} T_\omega d\omega = \left\{\mathcal{L}^{-1} T_\omega\right\}(Y)$.

$$\approx 2\bar{\alpha}\int_{c-i\infty}^{c+i\infty}\frac{d\omega}{2\pi i}\int_{\rho}dx_{12}\frac{\tilde{K}(x_{10},x_{12})T_{\omega}(Qx_{12})}{\omega+2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)} \tag{50}$$

Where we took the leading order of the term in curly braces in the last step. Using (49) and (50), we now see that in Mellin space, (46) takes the following form:

$$T_{\omega}(Qx_{10})=\bar{\alpha}\frac{v(Qx_{10})}{\omega+2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)}+2\bar{\alpha}\int dx_{12}\frac{\tilde{K}(x_{10},x_{12})T_{\omega}(Qx_{12})}{\omega+2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)} \tag{51}$$

Notice that if we redefine the kernel as

$$K(x_{10},x_{12}):=\tilde{K}(x_{10,}x_{12})-\delta(x_{10}-x_{12})\ln\left(\frac{x_{10}}{\rho}\right) \tag{52}$$

then (51) takes on a particularly simple form.

$$T_{\omega}(Qx_{10})=\bar{\alpha}\frac{v(Qx_{10})}{\omega+2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)}+2\bar{\alpha}\int dx_{12}\left[\frac{K(x_{10},x_{12})T_{\omega}(Qx_{12})}{\omega+2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)}\right. \tag{53}$$

$$\left.+\frac{\delta(x_{10}-x_{12})\ln\left(\frac{x_{10}}{\rho}\right)T_{\omega}(Qx_{12})}{\omega+2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)}\right]$$

$$T_{\omega}(Qx_{10})\left(\frac{\omega}{\omega+2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)}\right)=\bar{\alpha}\frac{v(Qx_{10})}{\omega+2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)}+2\bar{\alpha}\int dx_{12}\frac{K(x_{10},x_{12})T_{\omega}(Qx_{12})}{\omega+2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)} \tag{54}$$

$$\boxed{T_{\omega}(Qx_{10})=\frac{\bar{\alpha}}{\omega}v(Qx_{10})+\frac{2\bar{\alpha}}{\omega}\int dx_{12}K(x_{10},x_{12})T_{\omega}(x_{12}Q)} \tag{55}$$

This is, in fact, the celebrated BFKL equation. Let us now show that it yields the well known eigenvalue $\chi(\lambda)=\psi(1)-\frac{1}{2}\psi(1-\lambda/2)-\frac{1}{2}\psi(\lambda/2)$, with $\psi(x):=\frac{d}{dx}\ln\Gamma(x)$ being the digamma function and $\psi(1)=\gamma_e$ Euler's constant. Let us first manipulate the $\tilde{K}$ part of the kernel in (52). Recalling (32) and (33),

$$\begin{aligned}\tilde{K}(x_{10},x_{12}) &= \frac{1}{2\pi}\int_\rho^\infty \frac{x_{10}^2}{x_{12}^2 x_{20}^2} J(x_{21},x_{20})dx_{20} \\ &= \frac{2x_{10}^2}{\pi x_{12}}\int_\rho^\infty \frac{dx_{20}}{x_{20}} \frac{1}{\sqrt{[(x_{21}+x_{20})^2 - x_{10}^2]\,[x_{10}^2-(x_{21}-x_{20})^2]}} \end{aligned} \tag{56}$$

Bringing in an identity that relates transverse lengths and Bessel functions,

$$\frac{\pi}{2}\int_0^\infty bdbJ_0(bx_{01})J_0(bx_{20})J_0(bx_{12}) = \frac{1}{\sqrt{[(x_{21}+x_{20})^2 - x_{10}^2]\,[x_{10}^2-(x_{21}-x_{20})^2]}} \tag{57}$$

$$\tilde{K}(x_{10},x_{12}) = \frac{x_{10}^2}{x_{12}}\int_0^\infty bdbJ_0(bx_{01})J_0(bx_{12})\int_\rho^\infty \frac{dx_{20}}{x_{20}}J_0(bx_{20}) \tag{58}$$

Let us tackle the x_{20} integral:

$$\begin{aligned}\int_\rho^\infty \frac{dx_{20}}{x_{20}}J_0(bx_{20}) &= \lim_{y\to 0}\left[\int_0^\infty dx_{20}x^{y-1}J_0(bx_{20}) - \int_0^\rho dx_{20}x^{y-1}J_0(bx_{20})\right] \qquad (59)\\ &= \lim_{y\to 0} 2^{y-1}b^{-y}\frac{\Gamma(\frac{y}{2})}{\Gamma(1-\frac{y}{2})} - \frac{\rho^y}{y} \\ &= \lim_{y\to 0}\left(\frac{2}{b}\right)^y \frac{\Gamma(\frac{y}{2}+1)}{y\Gamma(1-\frac{y}{2})} - \frac{\rho^y}{y} \\ &= \lim_{y\to 0}\frac{\left(\frac{2}{b}\right)^y}{\Gamma(1-\frac{y}{2})}\frac{\Gamma(\frac{y}{2}+1)-\Gamma(1)}{y} + \frac{2^y b^{-y}}{\Gamma(1-\frac{y}{2})}\frac{\Gamma(1)}{y} - \frac{\rho^y}{y} \\ &= \lim_{y\to 0}\frac{\left(\frac{2}{b}\right)^y}{\Gamma(1-\frac{y}{2})}\frac{\Gamma(\frac{y}{2}+1)-\Gamma(1)}{y} + \frac{1}{\Gamma(1-\frac{y}{2})}\frac{\left(\frac{2}{b}\right)^y-\rho^y}{y} \\ &= \psi(1) - \ln\frac{b\rho}{2} \qquad (60)\end{aligned}$$

Note that in the above integral we used the standard formula $\Gamma(x) = \Gamma(x+1)/x$. In (59), we also used the approximation $J_0(bx_{20}) \approx 1$ in the second term (red), as its argument is bounded by ρ. The integral in the first term (blue) is given in Gradshteyn and Ryzhik [9], p. 668, 6.516-14.

Using this result, let us now evince the eigenvalue for the eigenfunction x_{12}^λ of the kernel K in (55). Although Mueller omits this derivation in [11] due to it being

"straightforward", it is still quite a bit of work to show. Given the importance of the BFKL eigenvalue, we will perform the full calculation. To do so, we will make use of the Taylor series for Bessel functions,

$$J_0(bx_{12}) = \sum_{m=0}^{\infty} \frac{(-1)^m}{(m!)^2} \left(\frac{bx_{12}}{2}\right)^{2m} \tag{61}$$

Other techniques used will be summarized below.

$\int dx_{12} K(x_{10}, x_{12}) x_{12}^{\lambda}$

$$= \int_{\rho}^{\infty} dx_{12} x_{12}^{\lambda} \left[\frac{x_{10}^2}{x_{12}} \int_0^{\infty} bdb J_0(bx_{01}) J_0(bx_{12}) \left(\psi(1) - \ln\frac{b\rho}{2}\right) - \delta(x_{10} - x_{12}) \ln\left(\frac{x_{10}}{\rho}\right)\right] \tag{62}$$

$$= \int_{\rho}^{\infty} dx_{12} x_{12}^{\lambda} \frac{x_{10}^2}{x_{12}} \left\{\int_0^{\infty} bdb J_0(bx_{01}) J_0(bx_{12}) \left(\psi(1) - \ln\frac{bx_{10}}{2}\right)\right\}$$

$$= \int_{\rho}^{\infty} dx_{12} x_{12}^{\lambda} \frac{x_{10}^2}{x_{12}} \sum_{m=0}^{\infty} \frac{(-1)^m}{(m!)^2} \int_0^{\infty} bdb \left(\psi(1) - \ln\frac{bx_{10}}{2}\right) \left(\frac{bx_{01}}{2}\right)^{2m} J_0(bx_{12})$$

$$= \int_{\rho}^{\infty} dx_{12} x_{12}^{\lambda} \frac{x_{10}^2}{x_{12}} \sum_{m=0}^{\infty} \frac{(-1)^m}{(m!)^2} \left(\frac{m!}{x_{12}^2 \Gamma(-m)} \psi(1) - \frac{\partial}{\partial(2m)}\right) \left[\left(\frac{x_{10}}{2}\right)^{2m} \int_0^{\infty} bdb J_0(bx_{12}) b^{2m}\right] \tag{63}$$

$$= \int_{\rho}^{\infty} dx_{12} x_{12}^{\lambda} \frac{x_{10}^2}{x_{12}^3} \sum_{m=0}^{\infty} \frac{(-1)^m}{(m!)^2} \left(\frac{m!}{\Gamma(-m)} \psi(1) - \frac{\partial}{\partial(2m)}\right) \left[2\left(\frac{x_{10}}{x_{12}}\right)^{2m} \frac{\Gamma(m+1)}{\Gamma(-m)}\right]$$

$$= \int_{\rho}^{\infty} dx_{12} x_{12}^{\lambda} \frac{x_{10}^2}{x_{12}^3} \sum_{m=0}^{\infty} \frac{(-1)^m}{(m!)^2} 2\left(\frac{x_{10}}{x_{12}}\right)^{2m} \left(\frac{m!}{\Gamma(-m)} \psi(1) - \ln\left(\frac{x_{10}}{x_{12}}\right) \frac{\Gamma(m+1)}{\Gamma(-m)} - \frac{\partial}{2\partial m} \frac{\Gamma(m+1)}{\Gamma(-m)}\right) \tag{64}$$

$$= \int_\rho^\infty dx_{12} x_{12}^\lambda \frac{x_{10}^2}{x_{12}^3} \sum_{m=0}^\infty \frac{(-1)^m}{(m!)^2} 2 \left(\frac{x_{10}}{x_{12}}\right)^{2m} \left(\frac{m!}{\Gamma(-m)}\psi(1) \right.$$

$$\left. - \frac{1}{2}\frac{\Gamma'(m+1)\Gamma(-m) + \Gamma'(-m)\Gamma(m+1)}{\Gamma^2(-m)}\right)$$

$$= \int_\rho^\infty dx_{12} x_{12}^\lambda \frac{x_{10}^2}{x_{12}^3} \sum_{m=0}^\infty \frac{(-1)^m}{(m!)^2} \frac{m!}{\Gamma(-m)} \left(\frac{x_{10}}{x_{12}}\right)^{2m} 2\left[\psi(1) - \frac{1}{2}\psi(m+1) - \frac{1}{2}\psi(-m)\right] \tag{65}$$

$$= \int_\rho^\infty dx_{12} x_{12}^\lambda \frac{x_{10}^2}{x_{12}^3} \sum_{m=0}^\infty \frac{(-1)^m}{m!} \frac{1}{\Gamma(-m)} \left(\frac{x_{10}}{x_{12}}\right)^{2m} 2\chi(-2m) \tag{66}$$

$$= \int_\rho^\infty dx_{12} x_{12}^\lambda \frac{x_{10}^2}{x_{12}^3} x_{01} \delta(x_{01} - x_{12}) \chi(\partial/\partial x_{12})$$

$$\boxed{\int dx_{12} K(x_{10}, x_{12}) x_{12}^\lambda = \chi(\lambda) x_{01}^\lambda} \qquad \square \tag{67}$$

In (62) (green) we used the orthogonality/closure relation for Bessel functions:

$$\int_0^\infty b db J_0(b x_{01}) J_0(b x_{12}) = \frac{1}{x_{12}} \delta(x_{10} - x_{12}) \tag{68}$$

In (63) (blue) we used the same Gradshteyn and Ryzhik integral as in (59). In (64) (red) and (66) (cyan) we used

$$\sum_{m=0}^\infty \frac{(-1)^m}{m!} \frac{1}{\Gamma(-m)} \left(\frac{x_{10}}{x_{12}}\right)^{2m} = \frac{x_{10}}{2} \delta(x_{10} - x_{12}) \tag{69}$$

causing the red term to drop out. In (63) (red) we used (68) and (69). Finally, in (65) (magenta) we defined

$$\chi(\lambda) := \psi(1) - \frac{1}{2}\psi\left(1 - \frac{\lambda}{2}\right) - \frac{1}{2}\psi\left(\frac{\lambda}{2}\right) \tag{70}$$

3.3 The Pomeron from BFKL

Now in possession of the BFKL eigenvalue equation (67), we may demonstrate the emergence of the Pomeron. Let us begin by inverse Mellin transforming the amplitude once again.

$$T_\omega(x_{12}, Q) = \int_{c-i\infty}^{c+i\infty} \frac{d\lambda}{2\pi i}(Qx_{12})^\lambda T_{\lambda\omega} \tag{71}$$

The BFKL equation (55) can be easily solved for $T_{\lambda\omega}$.

$$\int_{c-i\infty}^{c+i\infty} \frac{d\lambda}{2\pi i}\left[T_\omega(\omega - 2\bar{\alpha}\chi(\lambda))\right] = \int_{c-i\infty}^{c+i\infty} \frac{d\lambda}{2\pi i}\bar{\alpha}v_\lambda(Qx_{10}) \tag{72}$$

$$T_{\lambda\omega} = \frac{\bar{\alpha}v_\lambda}{\omega - 2\bar{\alpha}\chi(\lambda)} \tag{73}$$

where v_λ is the Mellin transform of $v(Qx_{10})$. Recalling (48), let us perform the inverse Laplace and inverse Mellin transforms on (73) to solve for the amplitude as a function of energy, where we expect pomeron behavior to manifest itself.

$$T(Y, Qx_{10}) = \int_{c-i\infty}^{c+i\infty} \frac{d\omega}{2\pi i}e^{\omega Y} \int_{c-i\infty}^{c+i\infty} \frac{d\lambda}{2\pi i}(Qx_{10})^\lambda \frac{\bar{\alpha}v_\lambda}{\omega - 2\bar{\alpha}\chi(\lambda)} \tag{74}$$

The ω integral is a simple residue.

$$T(Y, Qx_{10}) = \bar{\alpha}\int_{c-i\infty}^{c+i\infty} \frac{d\lambda}{2\pi i}v_\lambda e^{2\bar{\alpha}\chi(\lambda)Y + \lambda\ln(Qx_{10})} \tag{75}$$

Assuming that a) $\ln(Qx_{10}) \ll \bar{\alpha}Y$, or that the transverse momentum is not too large, and b) $v_\lambda$is a slowly varying function, the integral in (75) can be approximated by the saddle point method. This method evaluates the integral where the phase is approximately stationary. We can see where this occurs by examining the graph of $\chi(\lambda)$ shown on figure 12.

Let us use the expansion of $\chi(\lambda)$ around $\lambda = 1$ [40]:

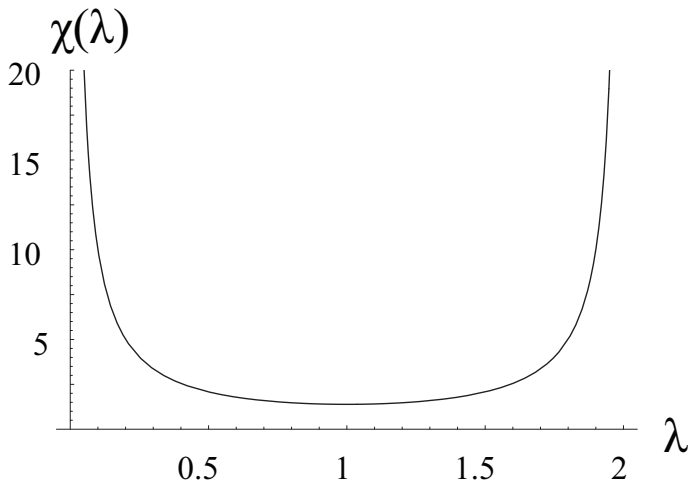

Figure 12: Graph of $\chi(\lambda)$ between $0 < \lambda < 2$. Note the saddle point at $\lambda = 1$.

$$\chi(\lambda) \approx 2\ln 2 + \frac{7}{4}\zeta(3)(\lambda-1)^2 \tag{76}$$

$$\chi'(\lambda) \approx \frac{7}{2}\zeta(3)(\lambda-1) \tag{77}$$

$$\chi''(\lambda) \approx \frac{7}{2}\zeta(3) \tag{78}$$

where $\zeta(x)$ is the Riemann zeta function. The saddle point approximation can be written as

$$\int_{c-i\infty}^{c+i\infty} d\lambda e^{f(\lambda)-\lambda\bar{x}} \approx \frac{1}{\sqrt{2\pi f''(\lambda_s)}} \exp\left(f(\lambda_s) - \lambda_s\bar{x} - \frac{\left[f'(\lambda_s)-\bar{x}\right]^2}{2f''(\lambda_s)}\right) \tag{79}$$

Applying this approximation to (75), along with (76), (77), and (78), we obtain

$$\begin{aligned} T(Y, Qx_{10}) &\approx \frac{\bar{\alpha}v_1}{\sqrt{14\bar{\alpha}\pi\zeta(3)Y}} \exp\left(4\bar{\alpha}\ln(2)Y - \ln(Qx_{10}) - \frac{\ln^2(Qx_{10})}{14\bar{\alpha}\zeta(3)Y}\right) \\ &= \frac{\bar{\alpha}v_1(Qx_{10})}{\sqrt{14\bar{\alpha}\pi\zeta(3)Y}} e^{(\alpha_{\mathbb{P}}-1)Y} \exp\left(-\frac{\ln^2(Qx_{10})}{14\bar{\alpha}\zeta(3)Y}\right) \end{aligned} \tag{80}$$

with

$$\boxed{\alpha_{\mathbb{P}} - 1 = 4\bar{\alpha}\ln(2)} \tag{81}$$

By (22), we see that BFKL evolution in the dipole picture indeed leads to the same hard pomeron behavior as in (16).

4 The BK Equation and Traveling Wave Solutions

4.1 Unitarity corrections to the BFKL equation; the BK equation

What are the consequences of a cross-sectional rise that goes like $e^{\alpha_{\mathbb{P}}-1}$ using (81)? Let us do a quick calculation: let $Q^2 \approx 10\,\text{GeV}$, a moderate value that does not violate the condition under (75). Using the well known formula of Gross, Politzer, and Wilczek for asymptotic freedom [53, 54],

$$\alpha_s(Q) = \frac{2\pi}{b_0 \ln(Q/\Lambda)}, \qquad b_0 = 11 - \frac{2}{3}n_f \tag{82}$$

Using $n_f = 3$ light quarks and $\Lambda = .2$ GeV, we obtain $\alpha_s = .178$. Then, with $N_c = 3$,

$$\alpha_{\mathbb{P}} - 1 = 4\bar{\alpha}\ln(2) = \frac{12\alpha_s}{\pi}\ln(2) \approx .47$$

Unlike for Reggeons (mesons ρ, ω, f_2, a_2,etc.) with a Regge trajectory intercept of[4] $\alpha(0) - 1 \approx -.45$, the BFKL pomeron, also called the hard pomeron, causes the cross-section to rise with s. This is actually necessary to fit available data, but with such a large power the Froissart-Martin bound [55] (a consequence of unitarity),

$$\sigma_{tot}(s) < \frac{\pi}{m_\pi^2}\ln^2\left(\frac{s}{s_0}\right) \tag{83}$$

is violated even within HERA's energy range. It is possible to introduce next to leading order (NLO) corrections to the BFKL equation that allow HERA data to be successfully fit [1], but even these are not enough to tame the eventual rise predicted by the LO BFKL equation[5]. A great deal of effort throughout the 90s went into formulating QCD evolution equations that preserve unitarity. This led to the B-JIMWLK equations [41, 56, 57], which were several different techniques: a functional renormalization group equation, an infinite hierarchy of coupled integro-differential equations, and a Langevin equation. In 1999, Kovchegov managed to considerably

[4] assuming degenerate trajectories for even and odd C-parity

[5] Interestingly, because the NLO correction is so substantial, Donnachie et. al. claim the perturbative ladder diagram calculation of the BFKL pomeron is suspect and that the correct value for the hard pomeron intercept provided by this calculation is probably a coincidence. See section 7.3 of [7] for details.

simplify Balitsky's equation using Mueller's dipole formulation, deriving what is now known as the BK equation. We will review the presentation of [39, 40] condensing and simplifying notation where possible.

Following [12][13], we will implement a dipole number density $n(x_{01}, Y, |\mathbf{b}|, x_1)$, which when convoluted with the photon dissociation wavefunction squared, $\Phi(z_1, x_{01})$, gives

$$N(x_1, Y) = \int d^2x_{01} \int_0^1 dz_1 \Phi(z_1, x_{01}) n(x_{01}, Y, x_1) \tag{84}$$

where $N(x_1, Y)$ is the propagator of the virtual photon through a target nucleus[6]. The BK equation is usually derived in the frame of the target with the evolution put into probe. We will see (84) obtains when we define $n(x_{01}, Y, x_1)$ by

$$\frac{1}{2\pi x_1^2} n_1(x_{01}, Y, |\mathbf{b}|, x_1) := \frac{\delta}{\delta u(\mathbf{x}_1)} \, Z(\mathbf{x}_{01}, Y, u)|_{u=1} \tag{85}$$

Likewise, we can define the dipole pair density

$$\frac{1}{2\pi x_1^2} \frac{1}{2\pi x_2^2} n_2(x_{01}, Y, x_1, x_2) = \frac{1}{2} \frac{\delta}{\delta u(\mathbf{x}_1)} \frac{\delta}{\delta u(\mathbf{x}_2)} \, Z(\mathbf{x}_{01}, Y, u)|_{u=1} \tag{86}$$

and generalizing to the group of k dipoles with sizes $x_1, \ldots, x_k$,

$$\prod_{i=1}^{k} \frac{1}{2\pi x_i^2} n_k(x_{01}, Y, x_1, \ldots, x_k) = \frac{1}{k!} \prod_{i=1}^{k} \frac{\delta}{\delta u(\mathbf{x}_i)} \, Z(\mathbf{x}_{01,} Y, u)|_{u=1} \tag{87}$$

The result of multiple functional differentiation in (87) is [39]

$n_i(x_{01}, Y, \mathbf{x}_1, \ldots, \mathbf{x}_k)$

$$\begin{aligned} &= \frac{\bar{\alpha}}{2\pi} \int_0^Y dy \exp\left[-2\bar{\alpha} \ln\left(\frac{x_{01}}{\rho}\right)(Y - y)\right] \int_\rho d^2\mathbf{x}'_2 \frac{x_{01}^2}{x_{02}^2 x_{12}^2} \\ &\times \left[2n_i(x_{02}, Y, \mathbf{x}_1, \ldots, \mathbf{x}_k) + \sum_{j+k=i} n_j(x_{02}, Y, \mathbf{x}_1, \ldots, \mathbf{x}_k) n_k(x_{12}, Y, \mathbf{x}_1, \ldots, \mathbf{x}_k)\right] \end{aligned} \tag{88}$$

[6] This is basically a rewriting of (22)

The total interaction cross-section is the sum of the interactions of each of the groups of k dipoles with the target. We can write this as

$$\begin{aligned} N(\mathbf{x}_{01},Y) &= \int \frac{d^2\mathbf{x}_1}{2\pi x_1^2} n_1(x_{01},Y,\mathbf{x}_1) \\ &\quad + \int \frac{d^2\mathbf{x}_1}{2\pi x_1^2}\frac{d^2\mathbf{x}_2}{2\pi x_2^2} n_2(x_{01},Y,\mathbf{x}_1,\mathbf{x}_2) + \ldots \\ &= \sum_{i=1}^{\infty} \int \frac{d^2\mathbf{x}_1}{2\pi x_1^2}\cdots\frac{d^2\mathbf{x}_i}{2\pi x_2^2} n_i(x_{01},Y,\mathbf{x}_1,\ldots,\mathbf{x}_i) \end{aligned} \tag{89}$$

Performing these operations on (88) yields

$$\begin{aligned} N(\mathbf{x}_{01},Y) &= \exp\left[-2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)Y\right] + \frac{\bar{\alpha}}{2\pi}\int_0^Y dy \exp\left[-2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)(Y-y)\right] \\ &\quad \times \int_\rho d^2\mathbf{x}_2 \frac{x_{01}^2}{x_{02}^2 x_{12}^2}\left[2N(\mathbf{x}_{02},y) - N(\mathbf{x}_{02},y)N(\mathbf{x}_{12},y)\right] \end{aligned} \tag{90}$$

Finally, taking the derivative of (90) with respect to Y,

$$\begin{aligned} \frac{\partial N(\mathbf{x}_{01},Y)}{\partial Y} &= -2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)\exp\left[-2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)Y\right] \\ &\quad \frac{\bar{\alpha}}{2\pi}\int_\rho d^2\mathbf{x}_2 \frac{x_{01}^2}{x_{02}^2 x_{12}^2}\left[2N(\mathbf{x}_{02},Y) - N(\mathbf{x}_{02},Y)N(\mathbf{x}_{12},Y)\right] \end{aligned} \tag{91}$$

Rewriting the the first term on the RHS to first order in $\bar{\alpha}$ as

$$-2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)\exp\left[-2\bar{\alpha}\ln\left(\frac{x_{10}}{\rho}\right)Y\right] = -\frac{\bar{\alpha}}{2\pi}\ln\left(\frac{x_{10}}{\rho}\right)\int_\rho d^2\mathbf{x}_2 4\pi\delta^2(\mathbf{x}_{01}-\mathbf{x}_{02})N(\mathbf{x}_{02},Y) \tag{92}$$

we can put (91) into a somewhat simpler form.

$$\frac{\partial N(\mathbf{x}_{01},Y)}{\partial Y} = \frac{\bar{\alpha}}{2\pi}\int_\rho d^2\mathbf{x}_2 \left\{\frac{x_{01}^2}{x_{02}^2 x_{12}^2}\left[2N(\mathbf{x}_{02},Y) - N(\mathbf{x}_{02},Y)N(\mathbf{x}_{12},Y)\right]\right. \tag{93}$$

$$-4\pi\delta^2(\mathbf{x}_{01}-\mathbf{x}_{02})\ln\left(\frac{x_{01}}{\rho}\right)N(\mathbf{x}_{02},Y)\Bigg\}$$

Notice that

$$\int_\rho d^2\mathbf{x}_2\frac{x_{01}^2}{x_{02}^2x_{12}^2}=2(2\pi)\int_\rho dx_{12}x_{12}\left(\frac{x_{01}^2}{x_{02}^2}\right)\frac{1}{x_{12}^2}=4\pi\int_\rho\frac{dx_{12}}{x_{12}}=4\pi\ln\left(\frac{x_{01}}{\rho}\right) \tag{94}$$

$$=\int_\rho d^2\mathbf{x}_2 4\pi\delta^2(\mathbf{x}_{01}-\mathbf{x}_{02})\ln\left(\frac{x_{01}}{\rho}\right)$$

where the factor of 2 after the first equality is due to evaluation at the collinear limit near both $\mathbf{x}_0$ and $\mathbf{x}_1$. If we take $x_{01}\approx x_{02}$, for instance, the second equality of (94) follows. Using (94), we can write the BK equation in another commonly used form (see [1][2]):

$$\frac{\partial N(\mathbf{x}_{01},Y)}{\partial Y}=\frac{\bar{\alpha}}{2\pi}\int_\rho d^2\mathbf{x}_2\frac{x_{01}^2}{x_{02}^2x_{12}^2}2N(\mathbf{x}_{02},Y)-N(\mathbf{x}_{01},Y)-N(\mathbf{x}_{02},Y)N(\mathbf{x}_{12},Y) \tag{95}$$

Aside from the nonlinear product $N(\mathbf{x}_{02},Y)N(\mathbf{x}_{12},Y)$, this equation is actually the same as the BFKL equation. We can crudely approximate when the solutions to the two equations diverge. Using the fact that the elementary dipole-dipole scattering amplitude is $T^{el}\sim\alpha^2$, the probability of two simultaneous scatterings is $\sim\alpha^4$, which is suppressed until the density of dipoles is $n\sim 1/\alpha^2$ (see (128,129) for details). At these densities, corrections provided by the nonlinear term are needed to stem the rise of the amplitude. Although the interpretation of this reduction in growth is not completely clear at present–be it due to gluon recombination, color swings, etc.–it must exist to preserve unitarity at high energies. In the t-channel picture, one can view the correction as replacing the single gluon ladder diagram with a "fan diagram" containing triple pomeron vertices, as in figure 13.

4.2 FKPP equation and reaction-diffusion dynamics

In this subsection we will show how the BK equation (95) encodes a branching diffusion of dipoles in the variable $\ln(1/r^2)$. The equation describing such diffusion is

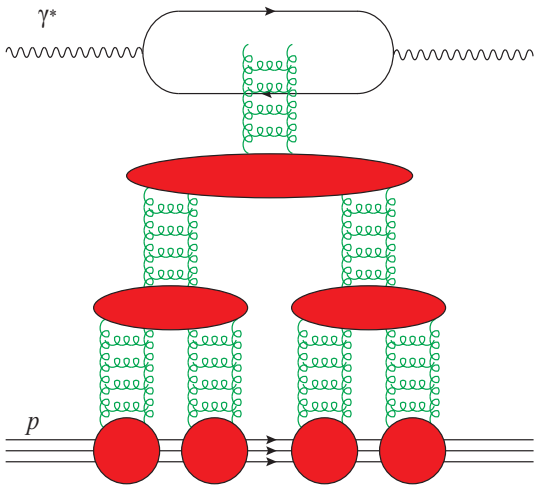

Figure 13: A fan diagram representing the BK equation in the t-channel.

called the Fisher-Kolmogorov-Petrovsky-Piscounoff (FKPP) equation, which is well known in statistical physics and is equivalent to the BK equation in the aptly named diffusion approximation. We will see the FKPP equation admits a traveling wave solution as dipoles diffuse to smaller sizes with increasing rapidity. The application of the FKPP equation to QCD evolution was first pointed out by Munier and Peschanski in a series of papers in 2003-4 [14, 15, 16].

Starting by Fourier transforming the BK equation (93) and using steps very similar to (62-67), we can rewrite the BK equation for momentum space $\tilde{N}(k,Y)$ using the BFKL eigenvalue $\chi_{Mueller}(\lambda)$[7] we found in (70) as [40]

$$\frac{\partial \tilde{N}(k,Y)}{\partial Y} = \bar{\alpha}\chi\left(-\frac{\partial}{\partial \ln k^2}\right)\tilde{N}(k,Y) - \bar{\alpha}\tilde{N}^2(k,Y) \tag{96}$$

Defining $L := \ln\left(k^2/\Lambda^2_{QCD}\right)$,

$$\frac{\partial \tilde{N}(k,Y)}{\partial Y} = \bar{\alpha}\chi\left(-\partial_L\right)\tilde{N}(k,Y) - \bar{\alpha}\tilde{N}^2(k,Y) \tag{97}$$

Using a series expansion of $\chi\left(-\partial_L\right)$ in the principle branch of the eigenvalue around a point $0 < \gamma_0 < 1$,

[7] N.B. We have made a trivial change to comply with more modern notation, $2\chi_{Mueller}(\lambda = 2(1-\gamma)) = \chi_{BFKL}(\gamma = 1 - \frac{\lambda}{2}) =: \chi(\gamma) = 2\psi(1) - \psi(1-\gamma) - \psi(\gamma)$ [44]. Thus the poles displayed in figure 12 are transformed like so: $\lambda = 0 \to \gamma = 1$ and $\lambda = 2 \to \gamma = 0$. Also, the saddle point at $\lambda_s = 1 \to \gamma_s = \frac{1}{2}$. For the remainder of this manuscript, we mean "χ_{BFKL}" when we write "χ".

$$\chi\left(-\partial_L\right) = \chi(\gamma_0)\mathbf{1} + \chi'(\gamma_0)(-\partial_L - \gamma_0\mathbf{1}) + \frac{1}{2}\chi''(\gamma_0)(-\partial_L - \gamma_0\mathbf{1})^2 + \frac{1}{6}\chi^{(3)}(\gamma_0)(-\partial_L - \gamma_0\mathbf{1})^3 + \ldots \tag{98}$$

The diffusion approximation is tantamount to keeping only up to second order terms in (98). Let us work with this truncated series and expand around $\gamma_0 = \frac{1}{2}$, as we did in the saddle point method used in 3.3.

$$\chi\left(-\partial_L\right) \approx \bar{\chi}\left(-\partial_L\right) := \chi\left(\frac{1}{2}\right) + \frac{\chi''\left(\frac{1}{2}\right)}{2}\left(\partial_L + \frac{1}{2}\right)^2 \tag{99}$$

If we make the following change of coordinates with $\omega := \chi\left(\frac{1}{2}\right)$, $D := \chi''(\frac{1}{2})$, and $\bar{\gamma} := 1 - \frac{1}{2}\sqrt{1 + 8\omega/D}$,

$$t := \frac{\bar{\alpha}D}{2}(1-\bar{\gamma})^2 Y \tag{100}$$

$$x := (1-\bar{\gamma})\left(L + \frac{\bar{\alpha}D}{2}Y\right) \tag{101}$$

$$u(t,x) := \frac{2}{D(1-\bar{\gamma})^2} N\left(\frac{2t}{\bar{\alpha}D(1-\bar{\gamma})^2}, \frac{x}{1-\bar{\gamma}} - \frac{t}{(1-\bar{\gamma})^2}\right) \tag{102}$$

then (97) with (99) becomes the FKPP equation:

$$\boxed{\partial_t u(t,x) = \partial_x^2 u(t,x) + u(t,x) - u^2(t,x)} \tag{103}$$

This equation is very well studied–see, for example, [45, 46] for comprehensive discussions. To quote from one of those references,

> The general goal of our discussion of front propagation into unstable states is to investigate the following front propagation problem: If initially a spatially extended system is in an unstable state everywhere except in some spatially localized region, what will be the large-time dynamical properties and speed of the nonlinear front which will propagate into the unstable state? Are there classes of initial conditions for which the front dynamics converges to some unique asymptotic front state? If so, what characterizes these initial conditions, and what can we say about the asymptotic front properties and the convergence to them? [45]

Reaction-diffusion	**QCD**
Occupation fraction u(t,x)	Scattering amplitude for the probe off a frozen realization of the target $T(k,Y)$, or $N(k,Y)$
Average occupation fraction $\langle u(t,x)\rangle$	Physical scattering amplitude $A = \langle T\rangle$
Space variable x, sometimes L	$\ln(k^2/\Lambda^2)$ or $\ln(1/r^2\Lambda^2)$
Time variable t	Rapidity $\bar{\alpha}Y$
Average maximum density of particles N	$1/\alpha^2$
Position of the front $X(t)$	Saturation scale $\ln(Q_s^2(Y)/\Lambda^2)$
Branching-diffusion kernel $\omega(-\partial_x)$, $(\omega(-\partial_x) = \partial_x^2 + 1$ for FKPP)	BFKL kernel $\chi(-\partial_{\ln k^2})$ or its equivalent in coordinate space

Table 1: A dictionary between reaction-diffusion and QCD variables. [1]

Let us turn our attention towards some of these issues. In short, an initial condition $u(0,x)$ will evolve into a traveling wave solution $u(t,x) = u(x-vt)$ with an asymptotic front velocity. Using the known result from FKPP analysis, [15]

$$u(t,x) \quad t \overset{\sim}{\to} \infty \quad w(x - 2t + \frac{3}{2}\ln t) \tag{104}$$

and assuming an exponential solution,

$$u(t,x) \sim \exp(x - 2t + \frac{3}{2}\ln t) \tag{105}$$

we may use the mappings (100), (101), and (102) to write

$$N(Y,k) \sim u(t,x) \sim \exp\left\{(1-\bar{\gamma})\left(L + \frac{\bar{\alpha}D}{2}Y\right) - 2\frac{\bar{\alpha}D}{2}(1-\bar{\gamma})^2 Y + \frac{3}{2}\ln\left[\frac{\bar{\alpha}D}{2}(1-\bar{\gamma})^2 Y\right]\right\} \tag{106}$$

$$= \exp(1-\bar{\gamma})\exp\left[L + \frac{\bar{\alpha}D}{2}Y - \bar{\alpha}D(1-\bar{\gamma})Y\right] Y^{\frac{3}{2(1-\bar{\gamma})}} \left(\frac{\bar{\alpha}D}{2}(1-\bar{\gamma})^2\right)^{\frac{3}{2(1-\bar{\gamma})}}$$

$$= k_0^{-2}k^2 \exp\left[-\bar{\alpha}D\left(\frac{1}{2} - \bar{\gamma}\right)Y\right] Y^{\frac{3}{2(1-\bar{\gamma})}}$$

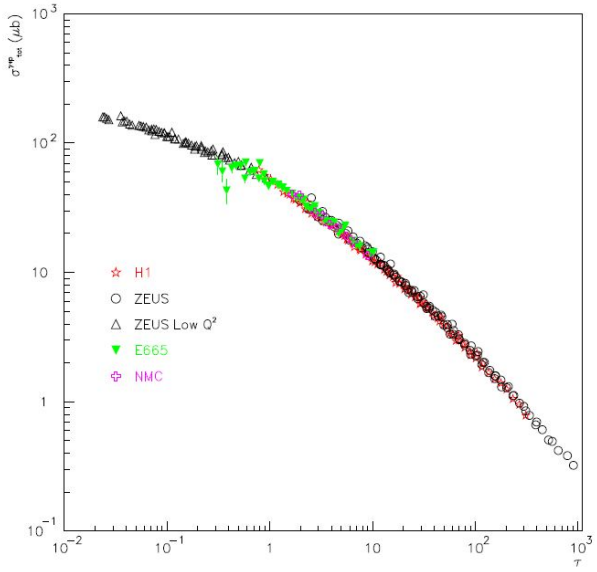

Figure 14: Geometric scaling data: the total cross section $\sigma_{tot}^{\gamma^* p \to X}$ as a function of $\tau := Q^2/Q_s^2(x)$ for $x < .01$. [38]

$$= \frac{k^2}{Q_s^2(Y)}, \qquad Q_s^2(Y) = k_0^2 Y^{-\frac{3}{2(1-\bar{\gamma})}} e^{\bar{\alpha} D(\frac{1}{2}-\bar{\gamma})Y} \tag{107}$$

where k_0^{-2} absorbs the constants. The result of these manipulations is to demonstrate that

$$\boxed{N(Y,k) = N\left(\frac{k^2}{Q_s^2(Y)}\right)} \tag{108}$$

which is the definition of geometric scaling, a feature strikingly revealed in the data, as shown in figure 14. Geometric scaling was known before Munier and Peschanski showed it was a consequence of the FKPP (see [35, 36]), but these authors framed the BK equation in the larger context of the universality class of the FKPP equation. In fact, the full BK equation (not using the diffusion approximation) and the NLO BFKL equation have both been shown to be a part of this universality class [1], meaning that all of these equations, details aside, exhibit branching diffusion with a saturation mechanism. This has been one of the pivotal discoveries in QCD over the last decade.

It is possible to analytically determine the velocity of the traveling wave predicted

by the FKPP equation. Because the wavefront mediates between the high density and low density regions in x, matching amplitudes at the two conditions allows us to determine a critical condition at the wavefront. This critical condition, in a certain interpretation, then yields the wavefront velocity.

First let us investigate the critical condition using a method explained in [47]. Starting from the BK equation (97), and using the Laplace transform,

$$N(k,\omega) = \int dY e^{-\omega Y} N(k,Y) \tag{109}$$

with a proposed ansatz [42]

$$N(k,\omega) = N(\omega) e^{[\gamma(\omega)-1]L} \tag{110}$$

where $L := \ln(k^2/\Lambda^2)$ as before and $\gamma(\omega)$ is the Mellin space argument of the BFKL eigenvalue (also called the anomalous dimension), we obtain

$$\omega e^{\omega Y} N(k,\omega) = \bar{\alpha}\chi(\gamma(\omega)) e^{\omega Y} N(k,\omega) - \bar{\alpha} \int_{c-i\infty}^{c+i\infty} \frac{d\omega'}{2\pi i} e^{(\omega+\omega')Y} N(k,\omega) N(k,\omega') \tag{111}$$

Shifting $\omega \to \omega - \omega'$ in the integral on the RHS,

$$\left[\omega - \bar{\alpha}\chi(\gamma(\omega))\right] N(\omega) e^{[\gamma(\omega)-1]L} = -\bar{\alpha} \int_{c-i\infty}^{c+i\infty} \frac{d\omega'}{2\pi i} N(\omega-\omega') N(\omega') e^{[\gamma(\omega-\omega')+\gamma(\omega')-2]L} \tag{112}$$

We may again use the saddle approximation (79) on the integral on the RHS, approximating around the choice $\omega' = \omega/2$ at which the derivative of the exponent vanishes. We obtain

$$\left[\omega - \bar{\alpha}\chi(\gamma(\omega))\right] N(\omega) e^{[\gamma(\omega)-1]L} = -\frac{\bar{\alpha}}{\sqrt{4\pi\gamma''(\omega/2)L}} N^2\left(\frac{\omega}{2}\right) e^{[2\gamma(\omega/2)-2]L} \tag{113}$$

In the region where the density is dilute, the nonlinear RHS is approximately zero, yielding

$$\text{Dilute Region :} \qquad \omega - \bar{\alpha}\chi(\gamma(\omega)) = 0 \tag{114}$$

On the other hand, we may match exponents in (113) in the saturation region to obtain a different condition.

$$\text{Saturation Region}: \qquad \gamma(\omega) = 2\gamma\left(\frac{\omega}{2}\right) - 1 \tag{115}$$

which is satisfied by

$$\gamma(\omega) = C\omega + 1 \tag{116}$$

for some constant C. We may solve for C using the derivative of (116) to obtain

$$\begin{aligned} \gamma(\omega) &= \gamma'(\omega)\omega + 1 \\ \gamma'(\omega) &= \frac{\gamma(\omega) - 1}{\omega} \end{aligned} \tag{117}$$

Taking the derivative of the dilute condition (114),

$$\bar{\alpha}\chi'(\gamma) = \frac{1}{\gamma'(\omega)} \tag{118}$$

Finally, we expect (118) to match with (117) at some critical value $\gamma_c = \gamma(\omega_c)$ at the wavefront where the dilute and saturation regions meet. Thus we obtain

$$\begin{aligned} \bar{\alpha}\chi'(\gamma_c) &= \frac{\omega_c}{\gamma_c - 1} \\ \chi'(\gamma_c) &= \frac{\chi(\gamma_c)}{\gamma_c - 1} \end{aligned} \tag{119}$$

where the second equality follows from evaluation of (114) at γ_c. (119) can also be rewritten using the symmetry of $\chi(\gamma)$ in its principle branch: $\chi(1-\gamma) = \chi(\gamma)$ and $\chi'(1-\gamma) = -\chi'(\gamma)$. Letting $1 - \gamma_c \to \gamma_c$,

$$\boxed{\chi'(\gamma_c) = \frac{\chi(\gamma_c)}{\gamma_c}} \tag{120}$$

This matching condition was actually first derived in the extensive 1983 Gribov, Levin, and Ryskin paper [48], but was rederived by Levin and Bartels in 1992 [42] with a more modern presentation.

More recently, in 2003 Munier and Peschanski [14] discovered a satisfying physical interpretation of the long known condition. Solving the linear part of the BK equation (97) as a wave packet in Mellin space,

$$N(k,Y) = \int_{c-i\infty}^{c+i\infty} \frac{d\gamma}{2\pi i} N_0(\gamma) e^{-\gamma L + \bar{\alpha}\chi(\gamma)Y} \tag{121}$$

we see that the phase velocity of a wave is

$$v_p = \frac{\chi(\gamma)}{\gamma} \tag{122}$$

and the group velocity is

$$v_g = \frac{d\chi(\gamma)}{d\gamma} \tag{123}$$

For the initial conditions relevant in QCD (a steeply falling function of L), FKPP analysis shows that the group velocity will equal the minimum phase velocity, which occurs at $\gamma = \gamma_c$.

$$v_g = v_p|_{min} = \frac{\chi(\gamma_c)}{\gamma_c} \tag{124}$$

$$\chi'(\gamma_c) = \frac{\chi(\gamma_c)}{\gamma_c} \tag{125}$$

which is the same as (120).

Before continuing, we will briefly address the nondeterministic nature of the evolution of the saturation scale. All that we have thus far discussed is deterministic and applies only to the mean field. However, because the formation of discrete dipoles ahead of the saturation front is a stochastic process, there will be some inherent dispersion among different "events", or realizations of BK evolution. As of currently, there has not been a rigorous proof of the behavior of this dispersion, but several numerical implementations have shown that

$$\sigma^2 \propto Y \tag{126}$$

There has been some progress in establishing this behavior using a "phenomenological" approach (see [18, 19]).

Part II

Model

5 Description of the Model: 2D, 2DR, and 2DSR

5.1 Overview

The object of our model is to implement Mueller's 2D branching kernel using a computer simulated Monte Carlo dipole generator. We expect the results to reproduce broad features of the FKPP traveling wave solution, in particular that the amplitude will behave like in figure 2 that we showed in the introduction, traveling with a fixed asymptotic velocity. Part of the motivation for this undertaking is to evaluate the following statement.

> Note that, though a full study with two transverse degrees of freedom would be of great interest, we believe that our one-dimensional picture grasps the important aspects of the problem and, based on universal properties of the reaction-diffusion systems, we expect our results to hold for full QCD. [24]

Will a 2D model reproduce the same universal properties as the 1D model? In what ways will the details be refined? We seek to answer these questions.

First let us define a model "event". An event begins with an initial set of dipoles of size $r_0 = 1$ randomly oriented and randomly distributed in impact parameter such that $|\mathbf{b}| < \frac{r_0}{2}$. Over the course of evolution in time[8], this initial dipole will have evolved into a multitude of smaller dipoles in each size index, exponentially at first but then tamed by a saturation mechanism. Each event consists of the movement of the saturation front ρ_s to successively smaller sizes over a specified time interval. Because we expect the solution to take the form of a traveling wave, the amplitude should be a function only of

$$T(\rho - \rho_s(Y)) = T\left(\frac{k^2}{Q_s^2(Y)}\right) \tag{127}$$

[8]Remember that $t \equiv Y$.

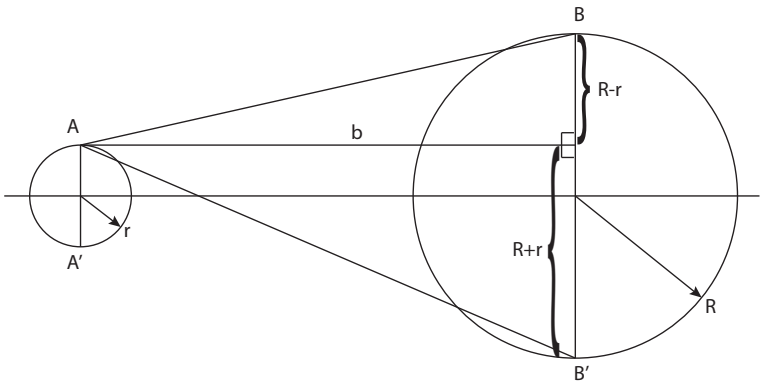

Figure 15: Geometry of a dipole-dipole scattering.

if $\rho \sim 1/r \sim k$. Thus, we see that $\rho_s(Y)$ plays the role of the saturation scale in the problem, and the traveling wave solution is equivalent to geometric scaling.

The amplitude can be calculated by making use of the following equation [17, 1].

$$T(y, \mathbf{x}_{01}) = \int \frac{d^2 z_0}{2\pi} \frac{d^2 z_1}{2\pi} T^{el}(\mathbf{x}_{01}, \mathbf{z}_{01}) n(y, \mathbf{z}_{01}) \tag{128}$$

where $n(y, \mathbf{z}_{01})$ is the dipole density, and the elementary scattering amplitude for a projectile dipole scattering off a target dipole is

$$T^{el}(\mathbf{x}_{01}, \mathbf{z}_{01}) = \frac{\pi^2 \alpha_s^2}{2} \ln^2 \frac{|\mathbf{x}_0 - \mathbf{z}_1|^2 |\mathbf{x}_1 - \mathbf{z}_0|^2}{|\mathbf{x}_0 - \mathbf{z}_0|^2 |\mathbf{x}_1 - \mathbf{z}_1|^2} \tag{129}$$

This formula represents the exchange of two gluons between a pair of dipoles, and as such is the square of the the single gluon potential between two dipoles in two dimensions [22]. It roughly counts the number of dipoles of similar size to x_{01}, which is convenient for computer implementation. Let us evince this feature. Given two dipoles of size $2r$ and $2R$, using the points shown on figure 15 T^{el} can be written

$$T^{el} = \frac{\pi^2 \alpha_s^2}{2} \ln^2 \frac{(AB')^2 (A'B)^2}{(AB)^2 (A'B')^2} \tag{130}$$

Case 1: $b \gg r, R$, leading order in R^2/b^2, rR/b^2, and r^2/b^2:

$$
\begin{aligned}
T^{el} &= \frac{\pi^2\alpha_s^2}{2}\ln^2\frac{\left[b^2+(R+r)^2\right]^2}{\left[b^2+(R-r)^2\right]^2} \\
&\approx \frac{\pi^2\alpha_s^2}{2}\ln^2\left\{\left[1+2\frac{(R+r)^2}{b^2}\right]\left[1-2\frac{(R-r)^2}{b^2}\right]\right\} \\
&\approx \frac{\pi^2\alpha_s^2}{2}\ln^2\left(1+\frac{8rR}{b^2}\right) \\
&\approx \frac{\pi^2\alpha_s^2}{2}\left(\frac{8rR}{b^2}\right)^2 = 32\pi^2\alpha_s^2\frac{(rR)^2}{b^4} \sim \frac{(rR)^2}{b^4}
\end{aligned} \tag{131}
$$

Case 2: $R > r, b$, leading order in r/R and b/R:

$$
\begin{aligned}
T^{el} &= \frac{\pi^2\alpha_s^2}{2}\ln^2\frac{\left[b^2+(R+r)^2\right]^2}{\left[b^2+(R-r)^2\right]^2} \\
&\approx \frac{\pi^2\alpha_s^2}{2}\ln^2\left[\left(1+\frac{4r}{R}\right)\left(1+\frac{4r}{R}\right)\right] \\
&\approx \frac{\pi^2\alpha_s^2}{2}\ln^2\left(1+\frac{8r}{R}\right) \\
&\approx \frac{\pi^2\alpha_s^2}{2}\left(\frac{8r}{R}\right)^2 = 32\pi^2\alpha_s^2\frac{r^2}{R^2} \sim \frac{r^2}{R^2}
\end{aligned} \tag{132}
$$

From (131) and (132), we see that dipoles which are far apart or which have very different sizes will not greatly contribute to (128).

5.2 Determination of splitting probabilities and lifetimes

Recall the transverse space kernel we derived in (30), which represents a classical branching probability[9]:

$$
\frac{dP_{x_{01}\to x_{02},x_{12}}}{dY} = \frac{x_{01}^2}{x_{12}^2 x_{02}^2}\frac{d^2\mathbf{x}_2}{2\pi} \tag{133}
$$

In order to derive an expression for the lifetime of a given size dipole and its probability of splitting into another size dipole, we will integrate (133) over $\mathbf{x}_2$. Changing coordinates to a polar coordinate system with origin $\mathbf{x}_1$ and expanding x_{02}^2 with the

[9]Note that "Y" in this model is actually rapidity scaled by $\bar{\alpha}$. I.e $\bar{\alpha}Y \to Y$ throughout Part II.

law of cosines,

$$\frac{dP_{x_{01}}}{dY} = 2x_{01}^2 \int_0^{2\pi} \frac{d\phi}{2\pi} \int_{r_{min}}^{r_{max}} \frac{dx_{12}}{x_{12}\left(x_{01}^2 + x_{12}^2 - 2x_{01}x_{12}\cos\phi\right)} \tag{134}$$

The lower limit r_{min} on the radial integral cuts off the collinear singularity, as we did in (44), whereas the upper limit r_{max} exists for the sake of computer implementation, as will become clear below. The left diagram in figure 16 shows the integration region around the point $\mathbf{x}_1$, with radial integration performed in such a way as to capture the collinear singularity around this point. This diagram depicts the parent dipole x_{01} splitting into two daughter dipoles, x_{12} and x_{02}. The placement of $\mathbf{x}_2$ determines both the lengths and positions of said daughters. Impact parameters (b_{01}, b_{02}, b_{12}) are defined to be the midpoint of the line segment joining the two endpoints of a given dipole. The result of this particular process will be two daughter dipoles with the parent removed.

Although it might be tempting to extend the integration region to the entire plane in such a polar coordinate system, there are two problems associated with doing so. First, using the logarithmic indexing shown in figure 16 left (which will be defined shortly), notice that if $x_{12} = x_{01}$ and if $\phi = 0$, measured with respect to the axis defined by x_{01}, then $x_{02} = 0$ and the integrand in (134) blows up. Of course, one could rotate the polar coordinate grid off of the singularity, but this brings us to our second point: symmetry dictates that we include the collinear singularity at $\mathbf{x}_0$ as well as $\mathbf{x}_1$. A simple method for doing so is to restrict the integration region to the vicinity of $\mathbf{x}_1$ and multiply by 2 to account for the symmetric probability distribution around $\mathbf{x}_0$. This accounts for the factor of 2 in (134).

So far we have only discussed how to capture the collinear singularity, but we must also include the infrared singularity when $x_{02}, x_{12} \gg x_{01}$ for our model to contain the proposed physics. Figure 16 right shows a scheme for covering most of the plane without overlap between the $\mathbf{x}_0$ and $\mathbf{x}_1$ regions. In practice we will divide the azimuthal range into 12 bins. Splittings of x_{01} to equal size daughter x_{12} are allowed in the azimuthal range $\frac{\pi}{3} \leq \phi < \frac{5\pi}{3}$, shaded in green, while all splittings to larger sizes are restricted to $\frac{\pi}{2} \leq \phi < \frac{3\pi}{2}$, shaded in yellow.

Continuing with the integral in (134) but switching to variable limits on ϕ,

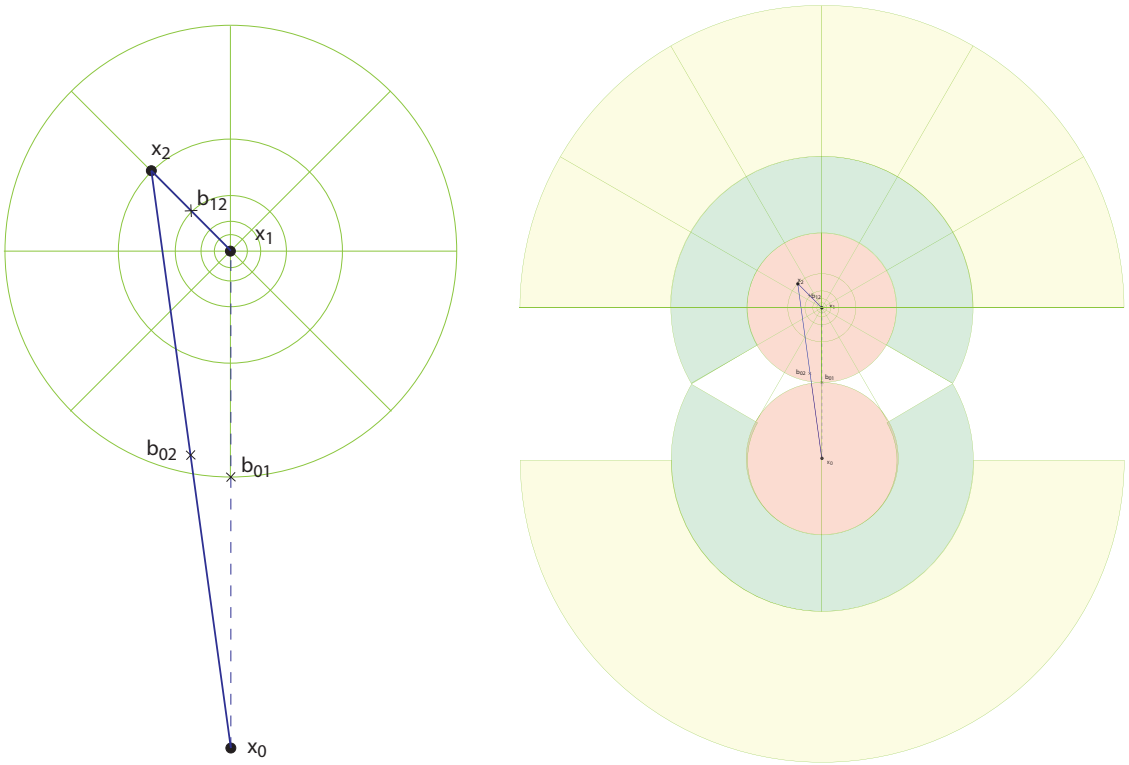

Figure 16: "Dartboard" diagrams indicating integration regions in (134). Left: Parent dipole x_{01} splitting into daughter dipoles x_{02} and x_{12}. The integration region is shown in the vicinity of $\mathbf{x}_1$. Right: The collinear region from the left figure is shaded in red, the equal size splitting region in green, and the infrared region in yellow. Only the first larger size splitting is shown for the infrared region, but the yellow region is understood to be an infinite radius section of a semicircle. The union of these three regions is mirrored for the region around $\mathbf{x}_0$.

$$\frac{dP_{x_{01}}}{dY} = \frac{1}{\pi}\int_{\phi_1}^{\phi_2} d\phi \int_{r_{min}}^{r_{max}} \frac{dx_{12}}{x_{12}\left(1+\frac{x_{12}^2}{x_{01}^2}-2\frac{x_{12}}{x_{01}}\cos\phi\right)} \tag{135}$$

$$= \frac{1}{\pi}\ln(B)\int_{\phi_1}^{\phi_2} d\phi \int_{\rho_{min}}^{\rho_{max}} \frac{d\rho}{1+B^{-2(\rho-\rho_x)}-2B^{-(\rho-\rho_x)}\cos\phi} \tag{136}$$

where logarithmic sizes are defined by $\rho := \log_B\left(\frac{1}{x_{12}}\right)$ and $\rho_x := \log_B\left(\frac{1}{x_{01}}\right)$. The base B determines the coarseness of the graining and will be taken to be 2 in the computer implementation of the model. Also, let $\rho_{min} := \log_B \frac{1}{r_{max}} = 0$ and $\rho_{max} := \log_B \frac{1}{r_{min}} = 50$ comprise the size limits on dipoles in our model[10]. We will approximate this integral as a Riemann sum for the purposes of computer implementation, with $\Delta\phi$ and $\Delta\rho$ chosen to be, respectively, $\frac{2\pi}{n}$ and 1. For $\rho_{min} \le \rho \le \rho_x$ the angular region will be restricted, as discussed above.

$$\begin{aligned}\frac{dP_{x_{01}}}{dY} \approx\ & \frac{1}{\pi}\ln(B)\sum_{\rho=\rho_{min}}^{\rho_x}\sum_{k=k_1}^{k_2}\frac{2\pi}{n}\frac{1}{1+B^{-2(\rho-\rho_x)}-2B^{-(\rho-\rho_x)}\cos\phi_k} \\ & +\frac{1}{\pi}\ln(B)\sum_{\rho=\rho_y+1}^{\rho_{max}-1}\sum_{k=0}^{n-1}\frac{2\pi}{n}\frac{1}{1+B^{-2(\rho-\rho_x)}-2B^{-(\rho-\rho_x)}\cos\phi_k}\end{aligned} \tag{137}$$

Letting $i = \rho_x$ and $j = \rho$,

$$= \sum_{j=\rho_{min}}^{i}\left.\frac{dP_{i\to j}}{dY}\right|_{j\le i} + \sum_{j=i+1}^{\rho_{max}-1}\left.\frac{dP_{i\to j}}{dY}\right|_{j>i} \tag{138}$$

since

$$\left.\frac{dP_{i\to j}}{dY}\right|_{j>i} = \frac{1}{\pi}\ln(B)\int_0^{2\pi} d\phi\int_j^{j+1}\frac{d\rho}{1+B^{-2(\rho-i)}-2B^{-(\rho-i)}\cos\phi}$$

$$= \frac{1}{\pi}\ln(B)\sum_{\rho=j}^{(j+1)-1}\sum_{k=0}^{n-1}\frac{2\pi}{n}\frac{1}{1+B^{-2(\rho-i)}-2B^{-(\rho-i)}\cos\phi_k}$$

[10] $\rho_{max} = 50$ is chosen due to the fact that 64-bit double precision binary floating-point numbers carry 1 bit of sign, 11 bits of exponent width, and 52 bits of significand precision. Thus, the maximum rounding error between two numbers, or machine epsilon, is 2^{-53}. ρ_{max} should be kept well below 53.

$$= \frac{1}{\pi} \ln(B) \sum_{k=0}^{n-1} \frac{2\pi}{n} \frac{1}{1 + B^{-2(j-i)} - 2B^{-(j-i)} \cos \phi_k} \tag{139}$$

and likewise,

$$\left. \frac{dP_{i \to j}}{dY} \right|_{j \leq i} = \frac{1}{\pi} \ln(B) \sum_{k=k_1}^{k_2} \frac{2\pi}{n} \frac{1}{1 + B^{-2(j-i)} - 2B^{-(j-i)} \cos \phi_k} \tag{140}$$

Thus, according to (138), the total probability for a dipole to split is the sum of the probabilities for it to split to any other size. For convenience, let us now define a probability splitting matrix $\mathcal{P}$ such that $\mathcal{P}_{ijk}$ is the kth term in the azimuthal sum of $\frac{dP_{i \to j}}{dY}$, i.e.

$$\mathcal{P}_{ijk} := \frac{1}{\pi} \ln(B) \frac{2\pi}{n} \frac{1}{1 + B^{-2(j-i)} - 2B^{-(j-i)} \cos \phi_k} \tag{141}$$

The $\mathcal{P}_{ijk}$ terms for which ϕ_k lies outside the azimuthal boundaries shown in figure 16 are set to 0. We can now write the total probability for the splitting of x_{01} (logarithmic size i) as

$$\frac{dP_i}{dY} = \sum_{j=\rho_{min}}^{\rho_{max}-1} \sum_{k=0}^{n-1} \mathcal{P}_{ijk} \tag{142}$$

and therefore, its "lifetime" in units of rapidity is

$$\tau_i = (dP_i/dY)^{-1} \tag{143}$$

The preceding forms the basis of our Monte Carlo calculation. During each step of the target's evolution in rapidity, the number of splittings of size i is determined according to

$$\#\ \text{splittings}_i = \frac{1}{\tau_i} \Delta Y \times (\#\ \text{dipoles of size i}) \tag{144}$$

We then randomly select this number of dipoles of size i, and for each selection choose a size j to split into using the discrete probability distribution

$$\frac{dP_{i\to j}}{dY} = \sum_{k=0}^{n-1} \mathcal{P}_{ijk} \tag{145}$$

This can be done, for example, by randomly choosing a number on the interval $[0,1]$ in the properly normalized cumulative distribution function of (145) and finding the corresponding ordinate. Similarly, we can randomly choose an azimuthal bin k to split into using the discrete probability distribution $\mathcal{P}_{ijk}$ for a given i and j[11].

5.3 Determination of $\mathbf{x}_2$

Once we have determined to which j and k a given dipole x_{01} will split, it is a simple matter to locate $\mathbf{x}_2$. If splitting from $\mathbf{x}_1$,

$$\mathbf{x}_2 = \mathbf{x}_1 - r_j \mathcal{R}(\frac{2\pi k}{n})\hat{\mathbf{x}}_{01} \tag{146}$$

$$\mathbf{x}_2 = \mathbf{x}_1 - r_j \mathcal{R}(\frac{2\pi k}{n})\frac{\mathbf{x}_1 - \mathbf{x}_0}{|\mathbf{x}_1 - \mathbf{x}_0|} \tag{147}$$

where $\mathcal{R}(\theta)$ is the standard rotation matrix,

$$\mathcal{R}(\theta) := \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{148}$$

By components,

$$\begin{aligned} x_{2x} &= x_{1x} - r_j(\cos\theta \hat{x}_{01,x} - \sin\theta \hat{x}_{01,y}) \\ x_{2y} &= x_{1y} - r_j(\sin\theta \hat{x}_{01,x} + \cos\theta \hat{x}_{01,y}) \end{aligned} \tag{149}$$

and

$$\mathbf{b}_{02} = \frac{\mathbf{x}_0 + \mathbf{x}_2}{2}, \qquad \mathbf{b}_{12} = \frac{\mathbf{x}_1 + \mathbf{x}_2}{2} \tag{150}$$

[11] In practice, to avoid creating ρ_{max}^2 discrete probability distributions for azimuth selection, we note that (141) depends on $j - i$, which is bounded between $-\rho_{max} \leq j - i \leq \rho_{max}$. Thus we only need to create $2\rho_{max} + 1$ discrete probability distributions.

If splitting from the $\mathbf{x}_0$ side, then (146) becomes

$$\mathbf{x}_2 = \mathbf{x}_0 + r_j \mathcal{R}(\frac{2\pi k}{n})\hat{\mathbf{x}}_{01} \tag{151}$$

mutatis mutandis.

5.4 Saturation veto and impact parameter cutoff veto

Limiting the number of dipoles in our model serves the dual purpose of satisfying unitarity constraints and ensuring computational efficiency. Toward this end, we will introduce two types of splitting vetoes into our model: the saturation veto and the impact parameter cutoff veto.

The former is based on the well known effect resulting from the BK equation, as discussed in 4.1. While the exact mechanism for saturation is not precisely known, be it a gluon recombination or shadowing effect, the results of our simulation should not strongly depend on the details. We will use the same condition as in [24, 25], which is that splittings that would generate daughters in regions already containing more than some N_{sat} number of dipoles will not be allowed. But how are we to count the number of such dipoles?

Observing figure 17, say we want to probe the number of dipoles of logarithmic size i in the vicinity of some impact parameter $\mathbf{b}_p$. We will count the number of dipoles whose impact parameters lie within an open ball around $\mathbf{b}_p$, $B_{r_i/2}(\mathbf{b}_p) := \{\mathbf{b} \in \mathbb{R}^2 | d(\mathbf{b}, \mathbf{b}_p) < r_i/2\}$[12]. Thus, in the figure the dipole with impact parameter $\mathbf{b}_1$ (shaded blue) is counted while that with $\mathbf{b}_2$ (shaded green) is not. However, even if this number of counted dipoles is less than N_{sat}, this does not guarantee that the saturation condition is not violated elsewhere. For example, say there are already N_{sat} dipoles with impact parameters very near $\mathbf{b}_2$. The addition of a dipole with impact parameter $\mathbf{b}_p$ will violate the saturation condition at some $\mathbf{b}_3 \in B_{r_i/2}(\mathbf{b}_p) \cup B_{r_i/2}(\mathbf{b}_2)$, even though fewer than N_{sat} dipoles have impact parameters within $B_{r_i/2}(\mathbf{b}_p)$. Thus, technically speaking we should check saturation at all $\mathbf{b} \in B_{r_i/2}(\mathbf{b}_p)$ to ensure the saturation condition is never violated, but in practice saturation checks are very computationally expensive to carry out. Our results show that if checks are carried out at $\mathbf{b}_p$, $\mathbf{b}_p - \frac{r_i}{2}\hat{\mathbf{b}}_p$, and $\mathbf{b}_p + \frac{r_i}{2}\hat{\mathbf{b}}_p$, amplitudes obey saturation, and these checks are

[12] Recall from the logarithmic size definition under (136) that $r_i = B^{-i}$.

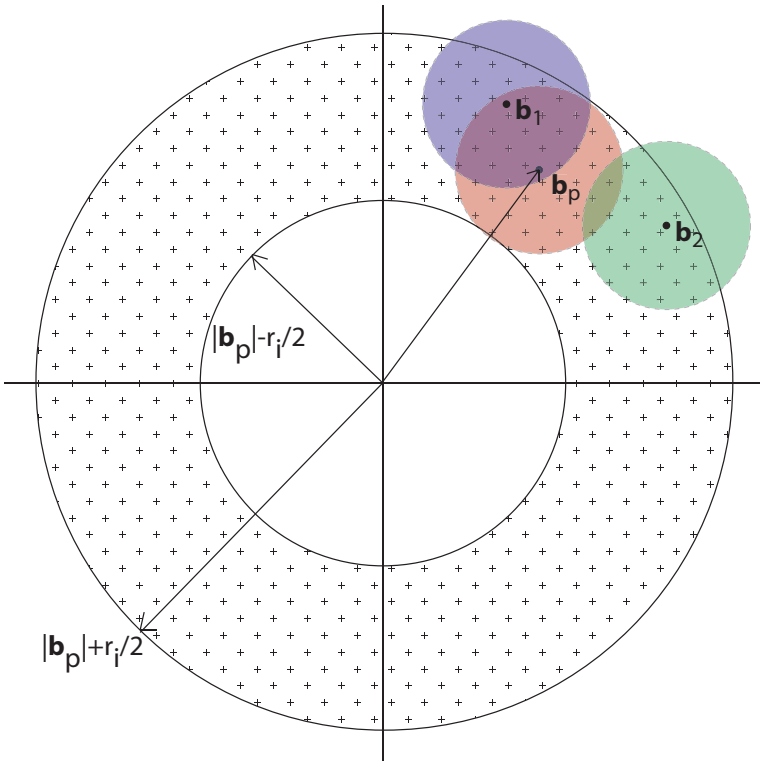

Figure 17: Two dipoles are shown with impact parameters $\mathbf{b}$ satisfying $|\mathbf{b}_p| - r_i/2 < |\mathbf{b}| < |\mathbf{b}_p| + r_i/2$. The dipole with impact parameter $\mathbf{b}_1$ (blue) is counted as being in the vicinity of $\mathbf{b}_p$ while that with $\mathbf{b}_2$ (green) is not. The crosshatched annulus is relevant to our search algorithm explained in 5.5.

ipso facto sufficient.

The other type of veto, which is a distance cutoff, is very easy to implement and necessary for computation in any reasonable length of time. If we choose a particular impact parameter $\mathbf{b}_p$ or set of impact parameters $\{\mathbf{b}_{p1}, \mathbf{b}_{p2}...\}$ at which to check the amplitude throughout the evolution of an event, most dipoles–especially very small sizes–will be too far from any of the $\mathbf{b}_p$ for them or their progeny to affect $T(\mathbf{b}_p)$. Therefore, we impose the same cutoff as in [24],

$$\frac{r_i}{|\mathbf{b} - \mathbf{b}_{pn}|} > \kappa \tag{152}$$

for some chosen value of κ in order to allow the splitting which creates a daughter dipole at $\mathbf{b}$ with size r_i. (152) must be satisfied for at least one of the $\{\mathbf{b}_{pn}\}$ for the splitting to be allowed; otherwise it is vetoed. We can see that this condition results in smaller dipoles being more strongly constrained to the probe location(s):

$$|\mathbf{b} - \mathbf{b}_{pn}| < \frac{r_i}{\kappa} \tag{153}$$

which is desirable, as there is no reason to keep track of the profusion of small dipoles that will not be observed. Typical values of κ we will be using are 10^{-1} and 10^{-2}. As long as κ is not close to 1, the asymptotic results of our model will not be greatly affected.

5.5 Data structure

2D evolution is much more computationally intensive than 1D due to the fact that a 2D transverse space can accommodate a far larger number of dipoles. Even given the veto constraints above, we must thoughtfully construct our data structure for computational efficiency. We can easily estimate the number of dipoles allowed for a given size i using (153). Say $\mathbf{b}_p = \mathbf{0}$, then dipoles of size i are constrained to a disk of radius

$$b_i < \frac{r_i}{\kappa} \tag{154}$$

The number of dipoles that can exist within this radius is approximately

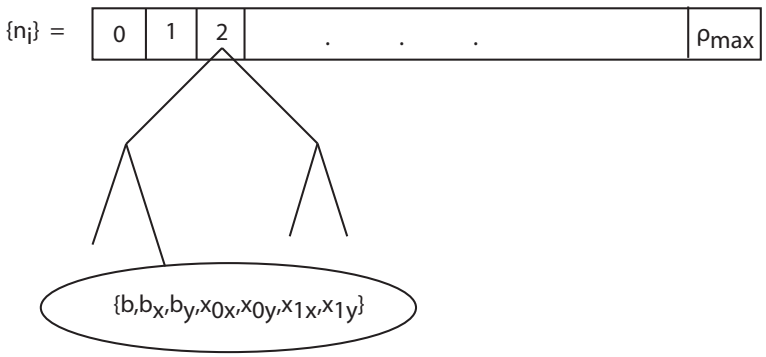

Figure 18: The data structure used to store dipoles. It is a vector with $\rho_{max}+1$ entries, each of which a red-black tree header node. Each red-black tree is ordered by magnitude of impact parameter.

$$N_i \approx N_{sat}\frac{\pi b_i^2}{\pi\left(\frac{r_i}{2}\right)^2} = \frac{4N_{sat}}{\kappa^2}$$

For typical values we will be using, $\kappa = 10^{-1}$ and $N_{sat} = 25$, $N_i \approx 10,000$. We have discovered that a 2D simulation becomes very computationally unwieldy when $N_i \gtrsim 10^5$. For this reason, $\kappa = 10^{-1}$ will be our standard choice for full 2D simulation.

The main data structure of the program will contain all of the dipoles created in the course of the target's evolution. It will consist of a vector $\{n_i\}_{i\epsilon\{0,1,2,\ldots,\rho_{max}\}}$, each index i of which represents all dipoles of logarithmic size i. The vector object type will be a binary red-black tree of nodes ordered by magnitude of impact parameter and that each contain the variables $\{b, b_x, b_y, x_{0x}, x_{0y}, x_{1x}, x_{1y}\}$. This is indicated schematically in figure 18.

Let us divert our attention to the red-black tree structure for each size index, which is crucial to the program's ability to quickly carry out saturation checks of the type described in 5.4. The conceptual basis for the red-black tree can be found in a number of references, for example its inventor's textbook, [50], but we will summarize the basic features here for the reader less familiar with data structures. Essentially, the red-black tree's purpose is to maintain the binary search tree's (BST) optimal $O(\log_2 N)$ search performance. It is one of several self-balancing tree algorithms available[13].

[13]The AVL tree is also sometimes used.

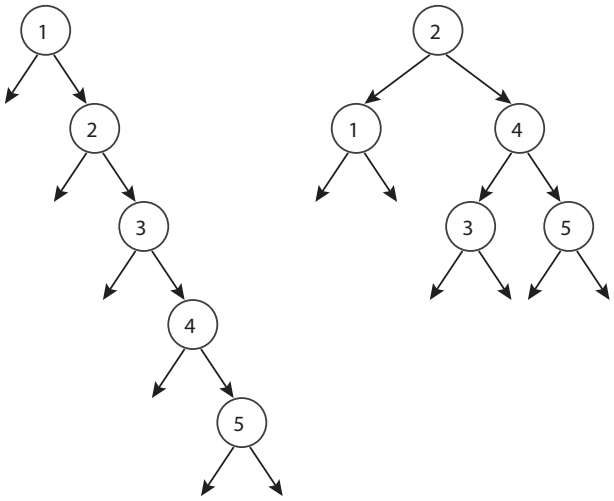

Figure 19: Left: A low efficiency BST with $O(N)$ search time. Right: A high efficiency BST with $O(\log_2 N)$ search time.

Consider the degenerate case of adding, in sequence, $1, 2, 3, 4, 5$ to a standard BST (figure 19). The insertion algorithm for a BST is to traverse the tree, going left if the node to be inserted is smaller than the current tree node, and right if it is greater. Thus, figure 19 left obtains with search time $O(N)$, as the BST degenerates into essentially a linked list in such cases. Figure 19 right obtains if we insert the sequence $2, 1, 4, 3, 5$, but we would like to achieve this efficient $O(\log_2 N)$ structure independent of insertion order. That is where the red-black tree comes into play.

A red-black tree's insertion and deletion algorithms ensure that its branches will remain roughly balanced at all times by leaving the following properties intact:

1. Each node is either red or black.
2. The root node is black.
3. Both children of every red node are black. If unsatisfied, there is said to be a "red violation".
4. Every path from root to leaf[14] contains the same number of black nodes. If unsatisfied, there is said to be a "black violation".

Such a tree satisfying these properties is shown in figure 20, as the reader may verify. Although the rebalancing algorithms are fairly detailed and refer to a number of

[14] The terminus of a path.

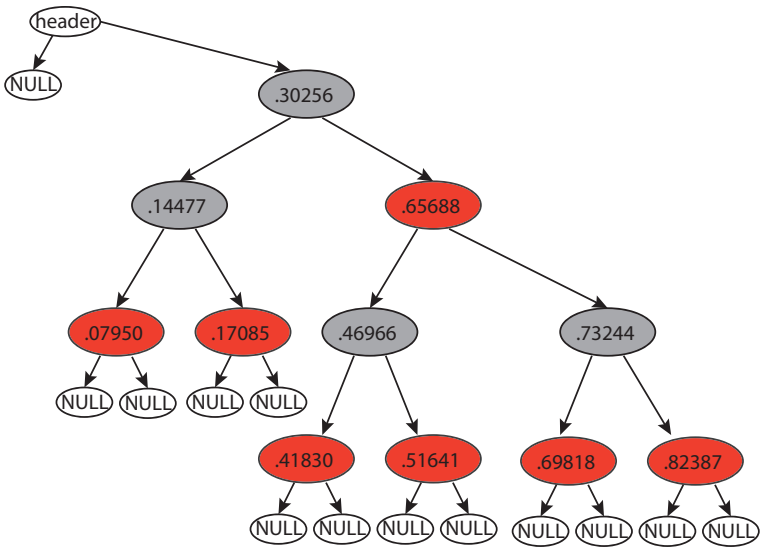

Figure 20: A sample red-black tree, ordered by magnitude of impact parameter

different cases, we will give one example to indicate the flavor of the operations required.

Say we are adding the node with impact parameter value ".93664". The red-black tree will now look like figure 21 upper. We can see that there is currently a red violation since the new node and its parent are both red. We cannot simply recolor the new node black, as this would lead to a black violation. Instead, we will recolor the new node's parent and grandparent, as shown in the diagram. Unfortunately, this causes another red violation. We cannot again recolor grandparent and great grandparent, as this would violate property 2. Thus, we can see that rotations are required for rebalancing. These rotations, along with recoloration and reattachment of appropriate subtrees are indicated in figure 21 middle. We end up with figure 21 bottom, which has the immediate visual appearance of being more balanced than 21 top.

Without going through all of the cases, suffice it to say that algorithms exist to maintain properties 1 through 4 during insertion and deletion of nodes. (The latter is especially tedious and is usually omitted from texts.) Several different types of algorithms actually exist to accomplish these tasks. The example given above is a type of "bottom-up" algorithm which recursively travels up the tree from the insertion point fixing mistakes on the way up. Another method involves nodes which have pointers from children to parents as well as from parents to children. However, both of these

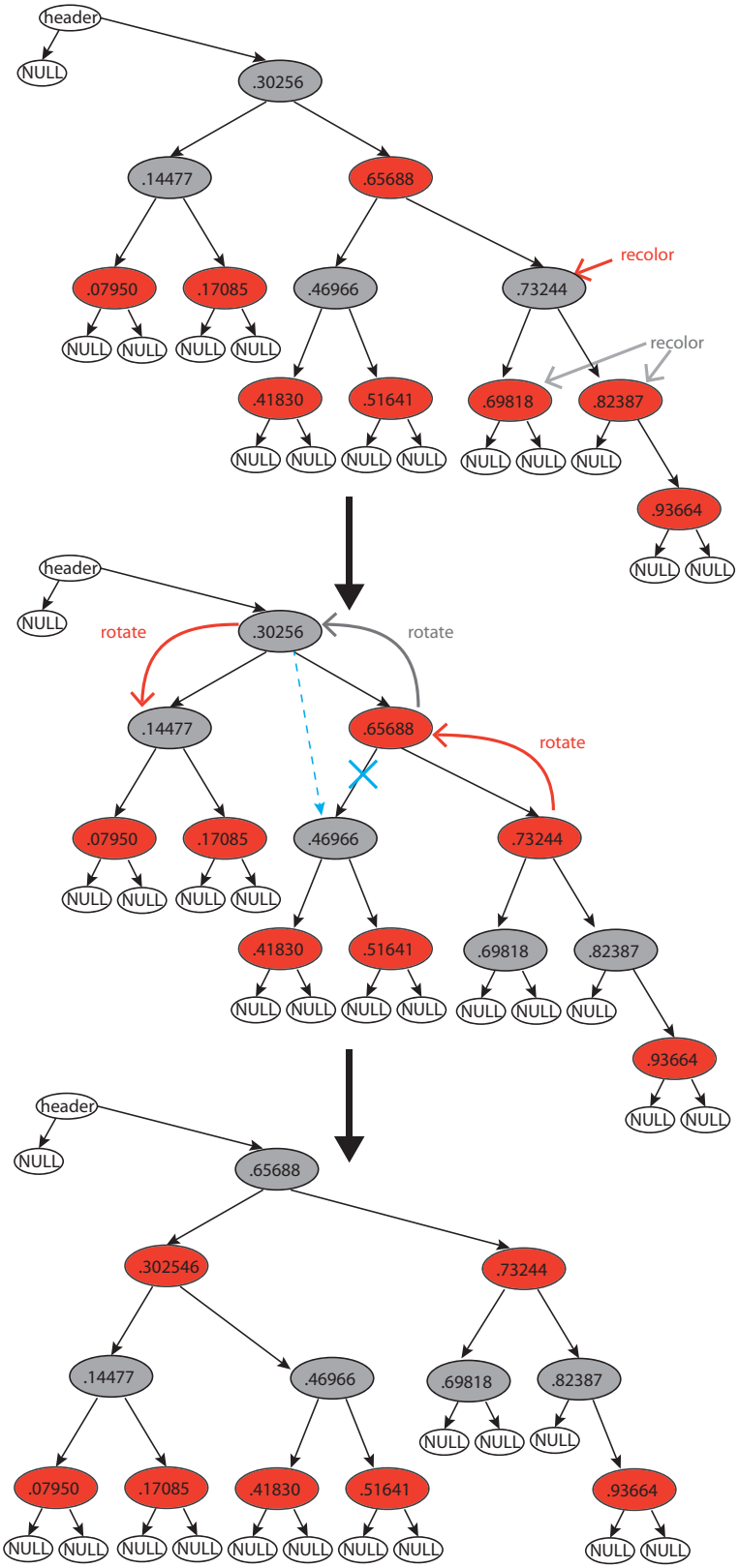

Figure 21: An example of red-black tree rebalancing after adding the node containing ".93664" on the far right.

methods appear somewhat inelegant when compared with "top-down" insertion. Top-down insertion is a nonrecursive method that makes changes on the way down the tree to the insertion point. Since it does useful work on the way down and does not have to traverse back up the tree, it is the most efficient method of implementing the red-black tree. It is surprisingly difficult to find these algorithms, but [51] provides a discussion of them.

Having the red-black tree data structure at our disposal allows us to quickly check $\{n_i\}$ for saturation vetoes, as explained in 5.4, and also to calculate the $T_i(\mathbf{b}_p)$, the amplitude at $\mathbf{b}_p$ for size i dipoles, at each step of the target's evolution. This is done by searching n_i, the ith red-black tree, for dipoles satisfying

$$\max(0, b_p - \frac{r_i}{2}) < b < b_p + \frac{r_i}{2} \tag{155}$$

This check is efficiently accomplished given the $O(\log_2 N)$ search performance of the red-black tree. Notice that (155) corresponds to the annulus in figure 17. Of course, we also need to check each dipole satisfying (155) to see whether

$$|\mathbf{b}_p - \mathbf{b}| < \frac{r_i}{2} \tag{156}$$

which is the number of dipoles with impact parameters within the open ball $B_{r_i/2}(\mathbf{b}_p)$, shaded red in figure 17. The sum of dipoles that satisfy (155) and (156) divided by N_{sat} yields $T_i(\mathbf{b}_p)$. Knowledge of $T_i(\mathbf{b}_p)$ for all i also allows us to calculate the saturation front, $\rho_s(\mathbf{b}_p, Y)$, which we will define as the smallest i such that $T_i(\mathbf{b}_p) < \frac{1}{2}$.

5.6 Parallel coding

Our C++ code was written using the OpenMP API for shared memory multiprocessing. Threading is controlled by the use of "`#pragma`" directives in the code, which stands for "pragmatic". These allow the C++ compiler to precisely control memory management and passing of parameters so as to offer machine and operating system-specific features while maintaining C++ compatibility. This platform independence allows the programmer to run the same code on machines of different number of cores while always utilizing the maximum advantage of multithreading on each machine.

Short data runs were performed on a typical home PC with 4 cores running at 2.67 GHz while longer runs up to 24 hours were performed on the Texas Advanced Com-

System Name:	Lonestar 4
Host Name:	lonestar.tacc.utexas.edu
Operating System:	Linux
Number of Processors:	22,656
Total Memory	44 TB
Peak Performance	302 TFLOPS
Total Disk:	276TB(local), 1000TB(global)

Table 2: TACC Lonestar 4 specifications

puter Center's (TACC) Lonestar 4 Dell Linux cluster. Without going into great detail, the basic specifications of this cluster are the following: [52]

The 22,656 cores are housed on 1,888 Dell PowerEdge M610 compute blades with 12 to a blade. Each blade has 2 Xeon 5680 series 3.33GHz hex-core processors. The user may submit jobs serially to each compute blade, which then multithreads the code onto 12 cores, providing essentially a 12-fold increase in the rate data production for our simulation. Multiple blades may be simultaneously harnessed, allowing further generation of data.

5.7 Pseudocode program

Most of what has not been described heretofore is merely nuts and bolts of programming, such as declarations, flow control statements, data output, and the like. The essential physics has all been described. For the reader interested in how the program works, we will give a pseudocode overview of the program flow. This description is for a single event–multiple events are simply repeated instances of a single event.

Program flow, single event

- main rapidity loop over Y:
  - loop over dipole size i:
    - calculate number of splittings for size i using lifetime, see (144)
    - loop over number of splittings, l:
      - choose a random dipole of size i to split

- monte carlo this dipole into size j dipole, see (145)
- monte carlo into kth angular bin
- randomly choose which side of dipole i to split, see (146) and (151)
- check if 2 daughter dipoles, $\mathbf{x}_{02}$ and $\mathbf{x}_{12}$, satisfy κ cutoff (152); if not, veto splitting
- check if 2 daughter dipoles, $\mathbf{x}_{02}$ and $\mathbf{x}_{12}$, violate saturation; if so, veto splitting
- if neither veto has been applied, insert $\mathbf{x}_{02}$ and $\mathbf{x}_{12}$ and remove $\mathbf{x}_{01}$ from the appropriate red-black trees in the data structure shown in figure 18

– output data for this ΔY step

5.8 First several steps of an event

To illustrate the operations of the program, let us visually inspect the first several splittings of a single initial dipole. The program randomly generates the following two splittings during the first ΔY step, as shown in figure 22.

First Splitting:

$x_{0x} = 0.744071,\ x_{1x} = -0.253879,\ x_{2x} = -0.269879$

$x_{0y} = -0.397142,\ x_{1y} = -0.333142,\ x_{2y} = -0.58263$

Second Splitting:

$x_{0x} = -0.253879,\ x_{1x} = -0.269879,\ x_{2x} = -0.305895$

$x_{0y} = -0.333142,\ x_{1y} = -0.58263,\ x_{2y} = -0.298492$

Both of these splittings occur in the collinear region, the first from $i = 0$ to $j = 2$ and the second from $i = 2$ to $j = 4$ (in logarithmic size). Notice that although the probability to split to a much smaller size is not improbable via (145), it is extremely improbable that a dipole created near the endpoints of its parent will pass the κ cutoff condition (152), which is to say it will likely be too far away from the region of interest to have any effect there. Over the course of the evolution of an event, smaller size dipoles will have found their way sufficiently near the probe location via other

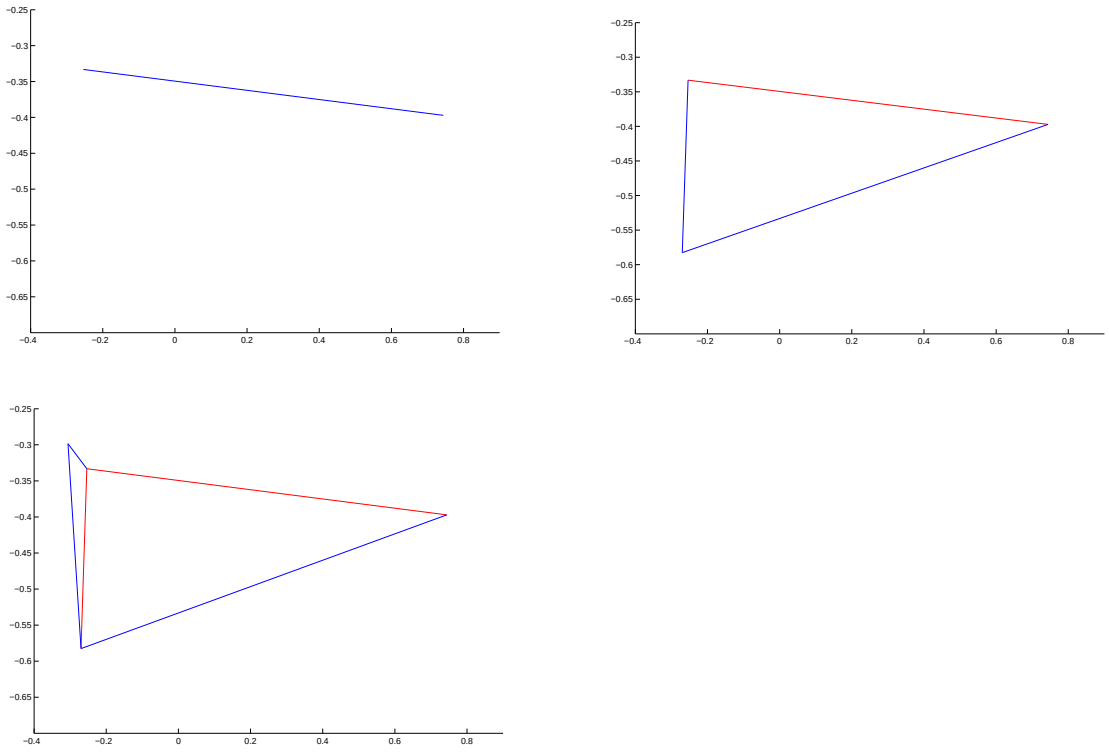

Figure 22: The first two splittings of an initial dipole shown in transverse space in clockwise progression. Removed parent dipoles are shown in red while extant dipoles are blue.

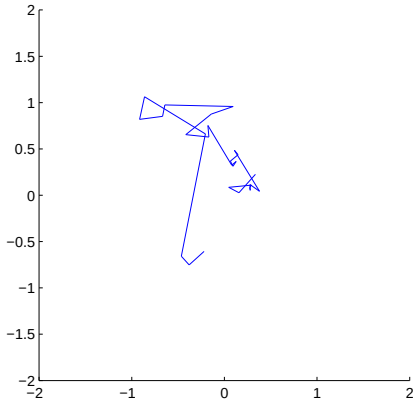

Figure 23: The evolution of a single dipole in transverse space at time $Y = 1$.

somewhat larger sizes to pass (152). In this sense the traveling wave moves smoothly from larger to smaller size dipoles over the evolution time.

After a longer period of time, the parent dipole will have branched into a multitude of various sized smaller dipoles, shown in figure (23). These daughter dipoles remain a connected graph, as the splitting rules imply.

Beginning with $N_{initial} = N_{sat} = 25$ dipoles and after sufficient time, a more fully evolved target is attained (figure 24).

5.9 2D Restricted (2DR)

It is desirable to have a way to check the results of our 2D calculation in the 1D limit in order to make contact with other work that has been done in 1D. To do so, we will employ the method illustrated in figure 25. The operation of the program is very similar to the 2D calculation, but with an added step before the veto conditions are checked. Recall that a logarithmic size and angle are chosen using discrete probability distributions, as described in 5.3. In the newly introduced step, the impact parameters $\mathbf{b_{12}}$ and $\mathbf{b}_{02}$ are projected onto the x-axis. If we were to simply project $\mathbf{x}_2$ onto the x-axis as well, this would have the effect of shortening the two projected dipoles x_{12} and x_{02}, especially in the case of an infrared splitting. Instead, we want to preserve the lengths x_{12} and x_{02}, which can be done by redefining the endpoints of the two daughters in the following way.

The endpoints of the dipole x'_{12} are given by

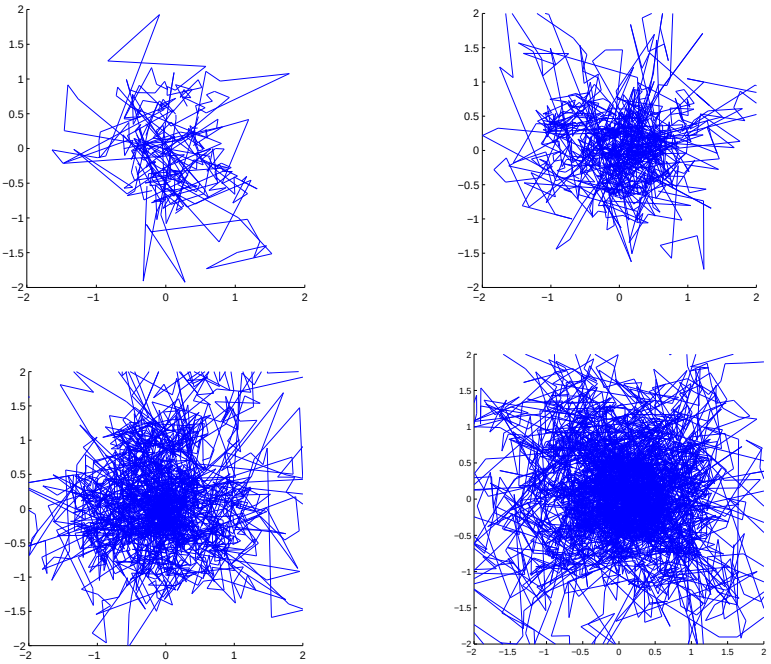

Figure 24: Going clockwise, target at time $Y = .5$, $Y = 1$, $Y = 1.5$, and $Y = 2$, all with $N_{initial} = N_{sat} = 25$ initial dipoles .

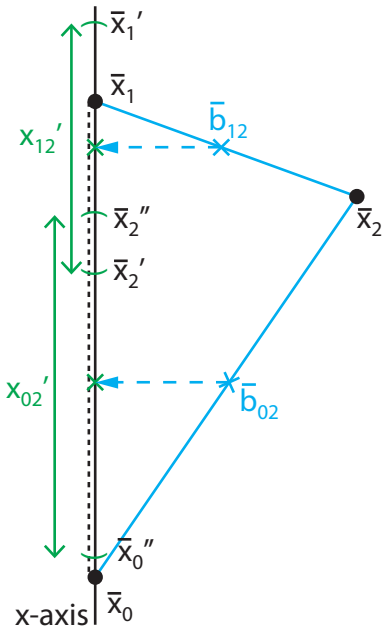

Figure 25: A method for reducing the full 2D calculation to 1D.

$$
\begin{aligned}
x'_{1x} &= b_{12x} + \frac{x_{12}}{2}\hat{x}_{01x} \\
x'_{2x} &= b_{12x} - \frac{x_{12}}{2}\hat{x}_{01x}
\end{aligned} \tag{157}
$$

and those of x'_{02} by

$$
\begin{aligned}
x''_{0x} &= b_{02x} - \frac{x_{02}}{2}\hat{x}_{01x} \\
x''_{2x} &= b_{02x} + \frac{x_{02}}{2}\hat{x}_{01x}
\end{aligned} \tag{158}
$$

In this scheme, we lose the shared endpoint between daughters, as $x'_{2x} \neq x''_{2x}$, but dipole sizes of the 2D model are preserved.

5.10 2D Semi-Restricted (2DSR)

In another variation of our model, this time we would like to be able to smoothly transition from the full 2D calculation to a 1D version of that calculation. The basic idea is to allow dipoles to evolve by spreading in the azimuth, but only within a certain defined strip width d around the x-axis. The shaded strip is shown in figure 26 left. Clearly the strip size must scale with the daughter dipole size if evolution is to be effectively constrained near the x-axis. We define d in the following way:

$$
d := \beta r_i \tag{159}
$$

where $r_i = \min(x_{02}, x_{12})$ is the size of the smaller daughter dipole and β is a factor that mediates the transition from 2D to 1D. If $\mathbf{x}_2$ lies within the strip then no projection takes place. If, on the other hand $|x_{2y}| > d$, then the projection

$$
x_{2y} \to x'_{2y} = sd, \qquad 0 < s < 1 \tag{160}
$$

shown in 26, left takes place, with s a random number in the interval above. In order to preserve the lengths of x_{02} and x_{12}, we slide $\mathbf{x}_0$ and $\mathbf{x}_1$ along the x-axis away from

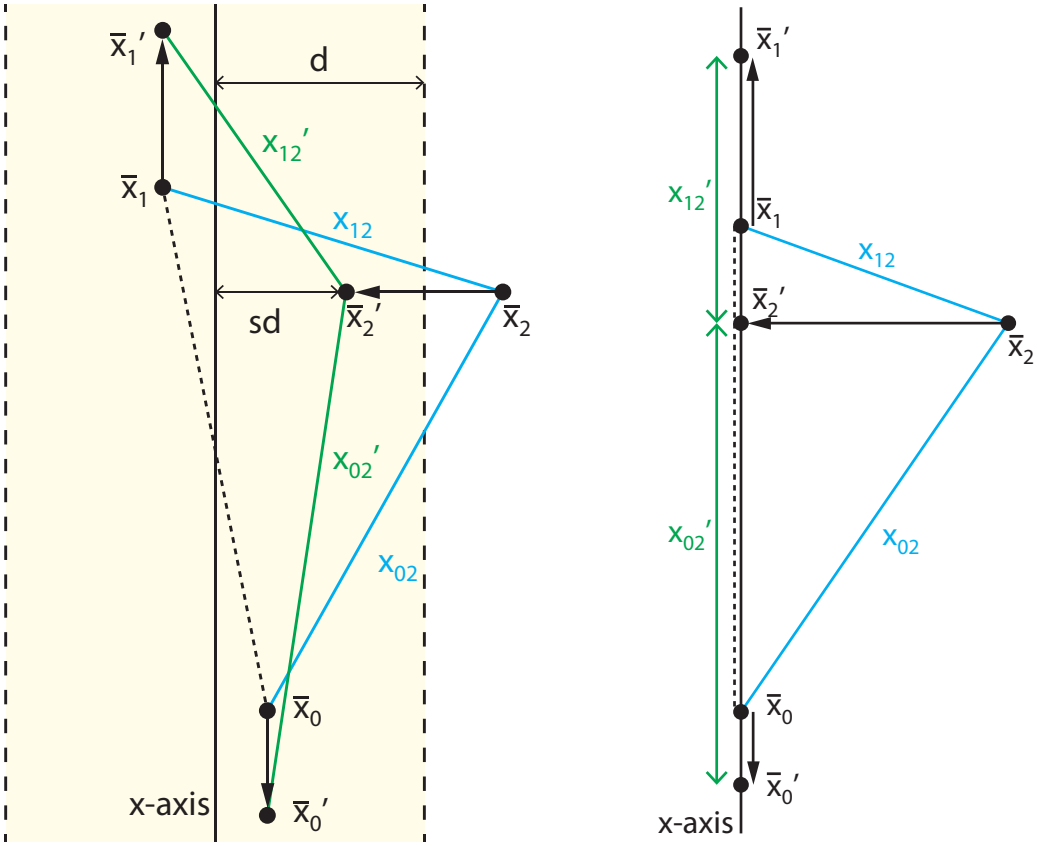

Figure 26: Left: Gluons emitted outside of a strip of width 2d around the x-axis are projected into the strip (shaded yellow) a distance sd away from the x-axis, where $0 < s < 1$. Right: The limit of 2DSR as the strip width $d \to 0$.

x_{2x}[15].

$$x'_{02} = x_{02}, \qquad x'_{12} = x_{12}$$

$$x'_{1x} = x'_{2x} \pm \sqrt{x_{12}^2 - (x_{1y} - x'_{2y})^2}$$

$$x'_{0x} = x'_{2x} \mp \sqrt{x_{02}^2 - (x_{0y} - x'_{2y})^2} \tag{161}$$

The end result of this scheme is that when $\beta \to \infty$ we recover the full 2D calculation, and when $\beta \to 0$ the calculation becomes 1D, as shown in figure 26 right. Note that this 1D limit is not exactly the same as the 2DR scheme, although the differences in the overall results between the two are minor.

[15] unless $x_{02}^2 < (x_{0y} - x'_{2y})^2$ or $x_{12}^2 < (x_{1y} - x'_{2y})^2$.

6 Results and Analysis

6.1 2D results

In presenting our results, we will display a number of the following quantities. Recall that the saturation front $\rho_s(Y, b)$ is a function of Y and b.

$$\frac{d\rho_s}{dY} = \frac{\langle \rho_s(Y + \Delta Y, 0) - \rho_s(Y, 0) \rangle}{\Delta Y} \tag{162}$$

$$\sigma^2 = \left\langle \rho_s^2(Y, 0) \right\rangle - \langle \rho_s(Y, 0) \rangle^2 \tag{163}$$

$$\mathrm{Cov}(b) := \mathrm{Cov}(\rho_s(Y, 0), \rho_s(Y, b)) = \langle \rho_s(Y, 0)\rho_s(Y, b) \rangle - \langle \rho_s(Y, 0) \rangle \langle \rho_s(Y, b) \rangle \tag{164}$$

And with these definitions,

$$\mathrm{Cov}(0) = \sigma^2 \tag{165}$$

as expected. Note that only ensemble averages are shown, and thus, individual events would have a more discrete appearance than the mean curves displayed on the amplitude plots. Also, individual events will be ahead of or behind the mean curves, the degree to which is indicated by the accompanying variance plots. Note that the attached C++ code only outputs the amplitude at various impact parameters and times. Additional data processing was handled in Matlab.

Figure 27 reveals the asymptotic wave speed to be about 3.5–much slower than the 1D models we will consider. Variance is proportional to Y after an initial wavefront formation time, as we expect from (126). The explanation for the the saturated region in 27 left having an amplitude slightly higher than 1 is the fact that we have only performed saturation checks at three points in transverse space when adding dipoles, as explained in 5.4. However, this slight excess has little effect on asymptotic values.

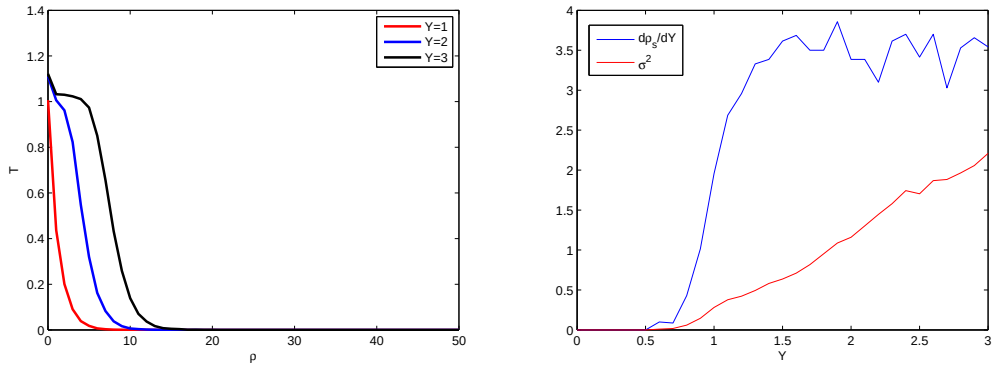

Figure 27: 2D Model: 700 events, $\kappa = 10^{-1}$

6.2 2DR results

Because the 2DR model is restricted to 1D, the number of dipoles allowed is severely curtailed when compared to 2D (see beginning of 5.5 for details). It is thereby much easier to gather high statistics in this version of the dipole model. With 5000 events in figure 28, wavefront velocity and variance curves are the smoothest of the data we present. The three point saturation check is also clearly more effective in 1D, as amplitudes are kept below $T(Y) = 1$ in the plots shown. Additionally, asymptotic wave velocity is seen to be much greater in 1D than in 2D, which we will discuss later. A comparison of figures 28 and 29 reveals that a change in κ has little effect on asymptotic velocity: $\langle d\rho_s/dY \rangle = 14.078$ for the former while $\langle d\rho_s/dY \rangle = 14.390$ for the latter[16]. It is slightly larger for the latter because $\kappa = 30^{-1}$ for this data allows dipoles to form within a radius three times greater (at a given i) than $\kappa = 10^{-1}$ for the former. Some of these additional dipoles that are farther from the probe location will be able to "walk in" through successive splittings. Further decreasing κ will have a diminishing effect on the wavefront velocity since the farther away a dipole is from the probe, the less likely it is have an effect there.

Figure 30 displays decorrelation of wavefronts at various impact parameters. This phenomenon is intuitively explained by considering the "resolution" of dipoles required to distinguish between two points. As long as the dipoles present in the simulation are larger than the separation between two impact parameters, these impact parameters are correlated and their covariance will rise over time. The points will decorrelate

[16]The velocity values were averaged over $Y = 1$ to $Y = 3$.

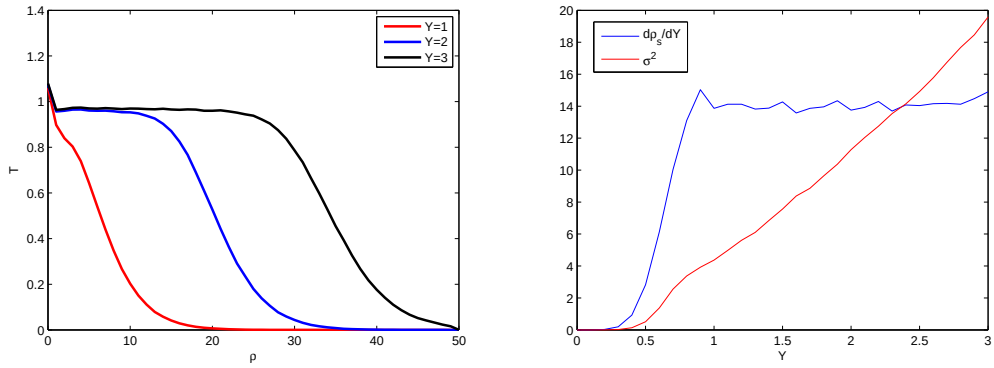

Figure 28: 2DR Model: 5000 events, $\kappa = 10^{-1}$

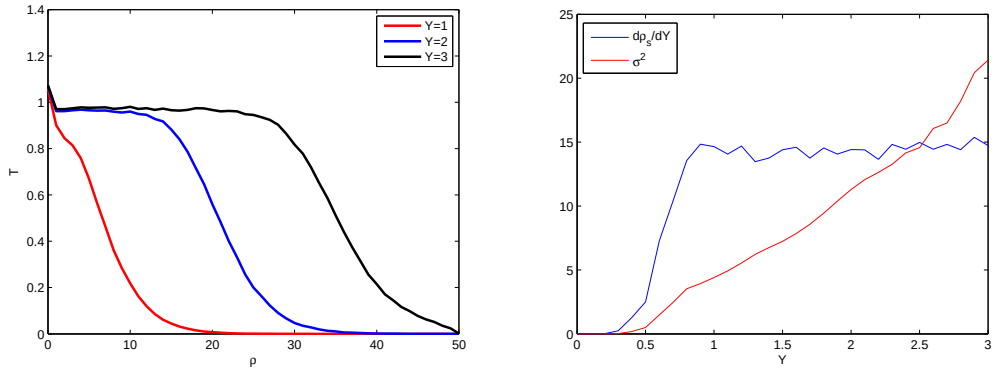

Figure 29: 2DR Model: 1000 events, $\kappa = 30^{-1}$

(their covariance will become constant) when the event has reached a fine enough resolution such that [25]

$$\Delta b \approx B^{-\rho_s(Y)} \tag{166}$$

Table 3 details the Y values at which various impact parameters decorrelate from $b = 0$. These Y values match well with figure 30.

6.3 2DSR results

The data from figure 31 interpolates between the 1D and 2D realizations of our model. As β increases, widening the projection strip, we see the essentially 1D results from the top row become the 2D results from the last row.

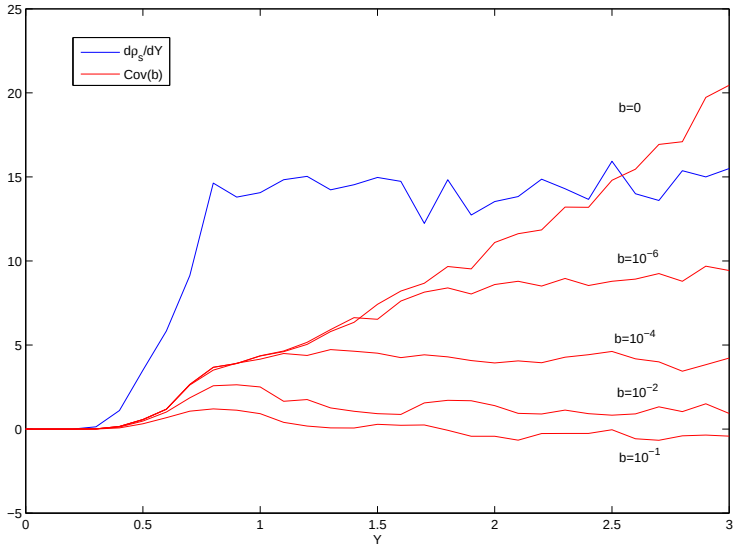

Figure 30: 2DR Model: 300 events, $\kappa = 10^{-1}$

Δb	ρ_s	Y_{decor}
10^{-6}	19.9	2
10^{-4}	13.3	1.5
10^{-2}	6.6	1
10^{-1}	3.3	.8

Table 3: Decorrelation data for impact parameters in figure 30 calculated using (166).

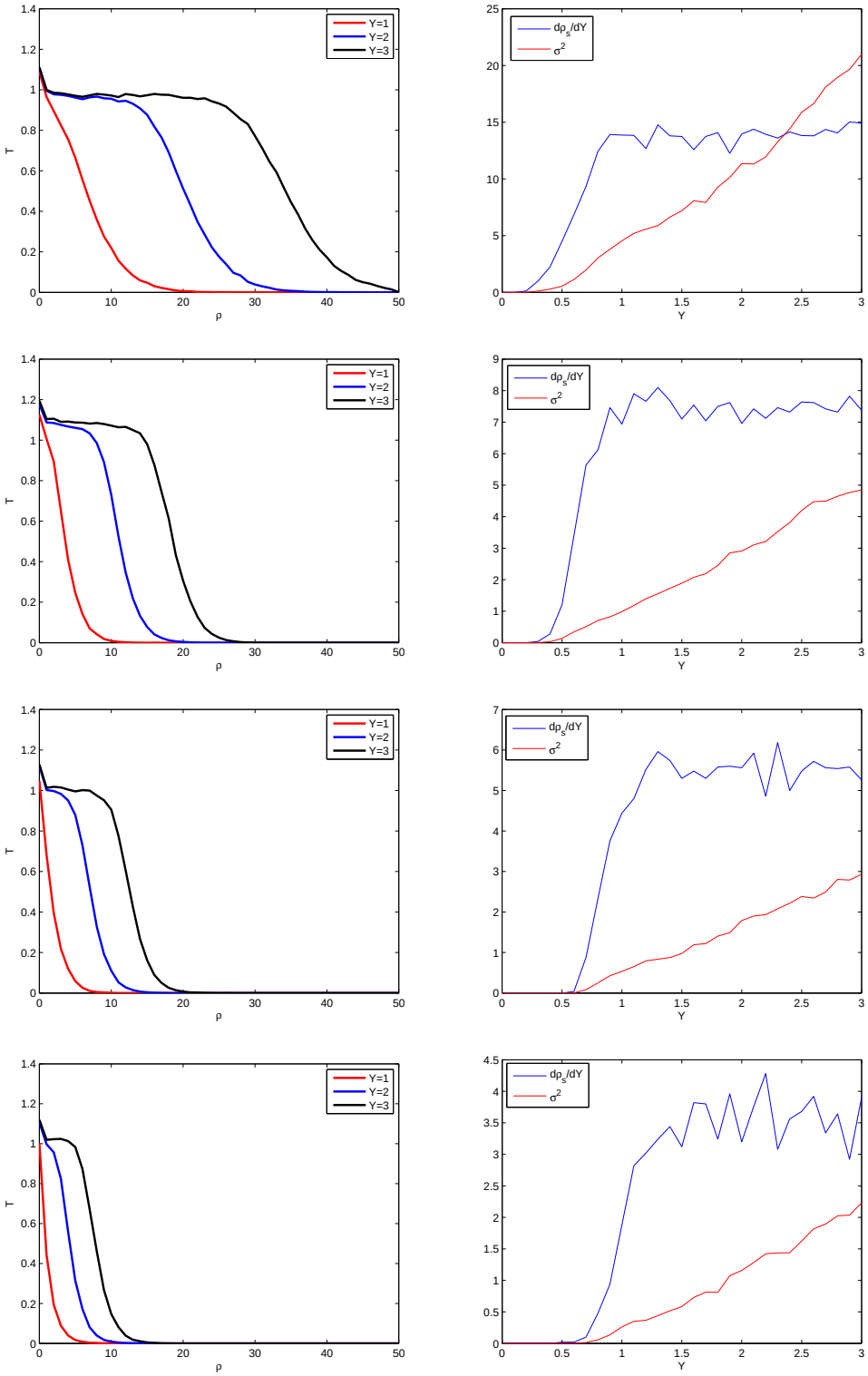

Figure 31: 2DSR Model: 1st row: $\beta = 0$; 2nd row: $\beta = 1$; 3rd row: $\beta = 3$; 4th row: $\beta = 100$. All data 500 events and $\kappa = 10^{-1}$.

6.4 Wavefront velocity analysis

6.4.1 1D Eigenvalue calculation

Splitting the kernel K_{ij} into infrared ($j < i$), collinear ($j > i$), and equal size ($j = i$) parts,

$$\begin{aligned}
\partial_Y n_i &= \sum_{j=\rho_{min}}^{\rho_{max}-1} K_{ij} n_j \\
&= \sum_{j=\rho_{min}}^{i} \left.\frac{dP_{i\to j}}{dY}\right|_{j<i} n_j + \sum_{j=i+1}^{\rho_{max}-1} \left.\frac{dP_{i\to j}}{dY}\right|_{j>i} n_j + \left.\frac{dP_{i\to j}}{dY}\right|_{j=i} n_j
\end{aligned} \tag{167}$$

The splitting probability is given by the BFKL kernel, transformed to logarithmic size index:

$$\begin{aligned}
\frac{dP_{i\to j}}{dY} &\equiv \frac{1}{\pi}\ln(B)\int_0^{2\pi} d\phi \int_j^{j+1} \frac{d\rho}{1 + B^{-2(\rho-i)} - 2B^{-(\rho-i)}\cos\phi} \\
&= \frac{1}{\pi}\ln(B)\sum_{\rho=j}^{(j+1)-1}\sum_{l=0}^{n-1}\frac{2\pi}{n}\frac{1}{1 + B^{-2(\rho-i)} - 2B^{-(\rho-i)}\cos\phi_l} \\
&= \frac{1}{\pi}\ln(B)\sum_{l=0}^{n-1}\frac{2\pi}{n}\frac{1}{1 + B^{-2(j-i)} - 2B^{-(j-i)}\cos\phi_l}
\end{aligned} \tag{168}$$

First we will handle the collinear term ($j > i$). Using the following approximation with $\zeta := \frac{x_{12}}{x_{01}} = B^{-(j-i)}$,

$$\begin{aligned}
\frac{1}{1 + \zeta^2 - 2\zeta\cos\phi} &= \sum_{m=0}^{\infty}(2\zeta\cos\phi - \zeta^2)^m \\
&= 1 + (2\zeta\cos\phi - \zeta^2) + 2\zeta^2(1 + \cos 2\phi) + O(\zeta^3) \\
&\to 1 + \zeta^2 + O(\zeta^3) \\
&\approx 1 + B^{-2(j-i)}
\end{aligned} \tag{169}$$

where the identity $(2\cos\phi)^2 = 2(1 + \cos 2\phi)$ was used in the second step, and the

integration of cosine terms set to 0 in the third, we can simplify the kernel. Using the eigenfunctions $\varphi_j = B^{j\gamma}$, and inserting a factor $\frac{r_j}{r_i} = B^{i-j} =: B^{-k}$ to reduce 1D to 0D fixed impact parameter (FIP),

$$\chi_{COL,1D}(\gamma)\varphi_i(\gamma) \approx \sum_{j>i} \frac{1}{\pi} \ln(B) \sum_{l=0}^{n-1} \frac{2\pi}{n} B^{-k}(1 + B^{-2(j-i)})\varphi_j(\gamma) \tag{170}$$

$$= 2\ln(B) \left(\sum_{k>0} B^{-k} B^{\gamma k}(1 + B^{-2k}) \right) \varphi_i(\gamma)$$

$$\chi_{COL,1D}(\gamma) = 2\ln(B) \left(\sum_{k>0} B^{k(\gamma-1)} + B^{k(\gamma-3)} \right)$$

$$= 2\ln(B) \left(\frac{1}{1 - B^{\gamma-1}} + \frac{1}{1 - B^{\gamma-3}} - 2 \right) \tag{171}$$

Restoring the Δ factors, taking the limit as $\Delta \to 0$, and using L'Hospital's Rule,

$$\lim_{\Delta \to 0} \chi_{COL}(\gamma) = \frac{2}{\gamma - 1} + \frac{2}{\gamma - 3} \tag{172}$$

we see that the $\gamma = 1$ singularity is present. Moving on to the infrared part ($j < i$),

$$\chi_{IR,1D}(\gamma)\varphi_i(\gamma) = \sum_{j=\rho_{min}}^{i} B^{-k} \left. \frac{dP_{i \to j}}{dY} \right|_{j<i} \varphi_j(\gamma) \tag{173}$$

Notice that since we are integrating semi-circles, we only sum over half of the azimuth.

$$= \sum_{k<0} \frac{1}{\pi} \ln(B) \sum_{l=0}^{half\ azimuth} \frac{2\pi}{n} B^{-k} \frac{1}{1 + B^{-2k} - 2B^{-k}\cos\phi_l} B^{j\gamma}$$

$$\approx \ln(B) \sum_{k<0} B^{-k} B^{2k} B^{j\gamma}$$

$$= \ln(B) \sum_{k<0} B^{k} B^{k\gamma} \varphi_i(\gamma)$$

$$\chi_{IR,1D}(\gamma) = \ln(B) \left(\frac{1}{1 - B^{-(\gamma+1)}} - 1 \right) \tag{174}$$

Finally, there is the $k = 0$ term, whose integral bounds come from figure 16,

$$\chi_{k=0,1D} = \frac{1}{\pi}\ln(B)\int_{\pi/3}^{5\pi/3}\frac{1}{1+B^0-2B^0\cos\phi} = \frac{\sqrt{3}}{\pi}\ln(B)$$

Adding the two parts (171) and (174),

$$\chi_{1D}(\gamma) = \chi_{COL,1D}(\gamma) + \chi_{IR,1D}(\gamma) + \chi_{k=0,1D}$$

$$= 2\ln(B)\left(\frac{1}{1-B^{\gamma-1}} + \frac{1}{1-B^{\gamma-3}} + \frac{1}{2(1-B^{-(\gamma+1)})} - \frac{5}{2} + \frac{\sqrt{3}}{2\pi}\right) \tag{175}$$

6.4.2 2D Eigenvalue calculation

Repeating all of the above steps but using instead the FIP factor of $\left(\frac{r_j}{r_i}\right)^2 = B^{-2k}$ to reduce 2D to 0D, (171) and (174) become

$$\chi_{COL,2D}(\gamma) = 2\ln(B)\left(\frac{1}{1-B^{\gamma-2}} + \frac{1}{1-B^{\gamma-4}} - 2\right) \tag{176}$$

$$\chi_{IR,2D}(\gamma) = \ln(B)\left(\frac{1}{1-B^{-\gamma}} - 1\right) \tag{177}$$

$$\chi_{2D}(\gamma) = 2\ln(B)\left(\frac{1}{1-B^{\gamma-2}} + \frac{1}{1-B^{\gamma-4}} + \frac{1}{2(1-B^{-\gamma})} - \frac{5}{2} + \frac{\sqrt{3}}{2\pi}\right) \tag{178}$$

6.4.3 Velocity calculations

Notice that for neither 1D nor 2D do we get both poles. (171) has the $\frac{1}{\gamma-1}$ pole, and (177) has the $\frac{1}{\gamma}$ pole. This is perhaps to be expected since the 1D FIP correction factor B^{-k} works well for the collinear sum in which dipoles remain more or less collinear. However, the 2D FIP factor B^{-2k} is better suited to the infrared sum since these kind of splittings allow the daughter dipoles to explore the azumithal range. We might consider using a "hybrid" eigenvalue function which has both of the correct poles,

$$\chi_{hybrid}(\gamma) \equiv \chi_{COL,1D} + \chi_{IR,2D} + \chi_{k=0}$$

$$= 2\ln(B)\left(\frac{1}{1-B^{\gamma-1}}+\frac{1}{1-B^{\gamma-3}}+\frac{1}{2(1-B^{-\gamma})}-\frac{5}{2}+\frac{\sqrt{3}}{2\pi}\right) \tag{179}$$

Using the eigenvalue functions (175), (178), and (179) and solving (120) using numerical methods, we obtain

$$V_{1D}=\frac{\chi'_{1D}(\gamma_c)}{\ln(2)}=12.67, \qquad \gamma_c=0.53$$

$$V_{2D}=\frac{\chi'_{2D}(\gamma_c)}{\ln(2)}=3.63, \qquad \gamma_c=1.19$$

$$V_{hybrid}=\frac{\chi'_{hybrid}(\gamma_c)}{\ln(2)}=15.35 \qquad \gamma_c=0.61 \tag{180}$$

Comparing these values to the data, we see the our analytical calculation for V_{2D} looks very accurate. Using the data shown in figure 27 we obtain $\langle d\rho_s/dY\rangle = 3.513$[17].The 2DR and 2DSR models suggest a value of $\langle d\rho_s/dY\rangle = 13.5$, which is still reasonably close to V_{1D}. We can also calculate the asymptotic velocity from the actual BFKL eigenvalue function, $\chi(\gamma)=2\psi(1)-\psi(1-\gamma)-\psi(\gamma)$. Using, for instance, [16]

$$\ln Q_s^2(Y)=\frac{\chi(\gamma_c)}{\gamma_c}Y-\frac{3}{2\gamma_c}\ln(Y/\bar{\alpha})-\frac{3}{\gamma_c^2}\sqrt{\frac{2\pi}{\chi''(\gamma_c)}}\frac{1}{\sqrt{Y}}+\mathcal{O}(1/Y) \tag{181}$$

the dominant term asymptotically yields

$$\begin{aligned}\frac{d\ln Q_s^2(Y)}{dY} &\approx \frac{\chi(\gamma_c)}{\gamma_c}\\ \frac{d\rho_s}{dY} &= \frac{1}{2\ln(2)}\chi'(\gamma_c)\approx 3.523\end{aligned} \tag{182}$$

which also compares well with our 2D model value.

[17]The velocity values were averaged over $Y=1.5$ to $Y=3$.

6.5 Conclusions

One facet of wave propagation we have noticed is the necessity of including both the infrared and collinear singularities of the branching kernel. Because the saturation front propagates to smaller dipole sizes over time, the collinear part of the kernel drives the wave forward in x while the infrared part "fills in" the unsaturated sizes behind the wavefront. Without the back-filling effect of the infrared term, the wave moves forward but is eventually damped out as the larger dipoles are replaced by dispersed smaller ones, and consequently, no stable wave shape asymptotically forms.

Comparing 1D and 2D data, it is seen that average wavefront velocities are considerably higher for the former. We have not seen a discussion of this effect in previous work, probably because no previous work has undertaken a model in two dimensions. One explanation why the saturation front progresses faster in 1D configuration space than in 2D is that dipoles spreading out in 2D transverse space with the same splitting probability as used in the 1D model become more dilute in comparison. As long as daughter dipoles are confined to a line, it is much more probable that each splitting will increase dipole density near the probe than in 2D. This reasoning still does not make the result *a priori* obvious, since one might imagine that the far more numerous dipoles in 2D could compensate for this dilution; however, it is seen that they do not. The analytical work in 6.4 gives some justification for this lower velocity.

We would like to consider the statement made in an earlier work,

> Note that, though a full study with two transverse degrees of freedom would be of great interest, we believe that our one-dimensional picture grasps the important aspects of the problem and, based on universal properties of the reaction-diffusion systems, we expect our results to hold for full QCD. [24]

Let us take stock of some of the assumptions made in the [24] model:

- Parent dipoles are retained throughout the evolution; collinear splitting rules create one small daughter dipole while the maintained parent approximates the other daughter. Infrared splitting probabilities are increased by a factor of 2 since only one daughter is created–the parent is still maintained.
- Dipole size is discrete: all dipoles have a size B^{-i} for some i.

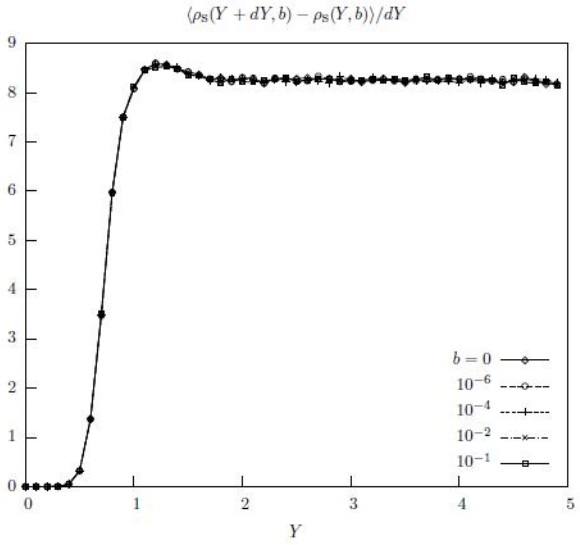

Figure 32: Average wavefront velocity, as shown in [24]

- The 2D kernel (133) is replaced by a 1D version,

$$\frac{dP}{dY} = \frac{x_{01}}{x_{02}x_{12}} dx_2$$

- The impact parameter of daughter dipoles is chosen using

$$b_j = b_i \pm \frac{r_i}{2} \pm \frac{r_j}{2} s, \qquad 0 < s < 1$$

We believe our model represents a more accurate calculation by avoiding all of these assumptions. The first assumption is obviated by replacement of the parent with two daughter dipoles in all cases. This assumption becomes questionable when the parent splits into a daughter of roughly the same logarithmic size, for example when an $i = 0$ parent splits into two $j = 1$ daughters. In this case it is not accurate to maintain the parent since neither of the daughters are the same size. In fact, most allowed splittings are of this nature since a splitting where $j - i$ is large is unlikely to pass the κ cutoff condition (152). Possibly this difference accounts for our 1D wavefront velocity being higher than that of [24] (shown in figure 32), as sizes can be driven downwards faster when parents are removed and replaced by two smaller dipoles. Assumption 2 is not present in our model, since splittings like that shown in figure 16 left create dipoles that are not equal to B^{-i} for any i. Assumption 3 reasonable in the collinear and infrared limits, but again, if $|j - i|$ is small then it is not accurate.

Assumption 4 is not necessary in our model because the splitting kernel determines the impact parameters of all daughter dipoles.

In summary, the splittings most relevant to driving the saturation front forward are those between similarly sized parent and daughter dipoles. Thus, it is important to handle these splittings accurately. We believe our model succeeds in this respect, and that it is therefore a more accurate model of dipole evolution than those previously wrought.

7 Final Summary

In this final chapter, we will more or less repeat was has already been said as concisely as possible. In chapter 2 we saw that a Regge trajectory with intercept greater than 1 called the Pomeron was needed to explain the rise of hadronic cross-sections. We then gave an account of how a pQCD calculation in the form of an infinite gluon ladder diagram could account for such a trajectory. In chapter 3 we introduced the dipole formulation for calculating cross-sections such as $\gamma^* p \to X$. In this picture, the virtual photon dissociates into a quark-antiquark pair which then interacts with the initial state hadron. Using this picture, Mueller showed that evolution of the target with increasing energy could be viewed as a highly occupied Fock state called an onium. Colorless dipoles comprise these states, which form due to soft gluon emissions. Using the wavefunction for the onium state, Mueller derived an integral equation which was equivalent to the BFKL equation found via the gluon ladder diagram, albeit the result of a much simpler calculation.

Although the BFKL equation correctly predicts dipole density growth in the dilute regime, in chapter 4 we explain that the eventual violation of unitarity with increasing s necessitates a nonlinear growth taming term. This is provided by the BK equation, which adds a $-N^2$ term to the evolution equation, providing the desired effect. It was later shown by Munier and Peschanski that the BK equation belongs to the universality class of the FKPP equation, familiar from reaction-diffusion dynamics. This conceptual framework allowed the phenomenon of geometric scaling to be viewed as a traveling wave whose front is the logarithm of the saturation scale. This front moves with a group velocity equal to the minimum phase velocity of a wave packet in Mellin space, a condition that can be found by matching conditions in the dilute and saturation regions.

In Part II, we move on to describe a model based on the classical branching kernel of the BFKL equation and a saturation mechanism. Both the collinear and infrared parts of the kernel are taken into account. Saturation is checked by the program efficiently through the use of the red-black tree data structure. A full 2D implementation of the model as well as a 1D variant and a smooth interpolation between 1D and 2D are introduced. Data on wavefront asymptotics and correlations in impact parameter are presented and contrasted with an earlier work based on a 1D model. Finally, analytical calculations of the wavefront asymptotic velocity are compared with the

data.

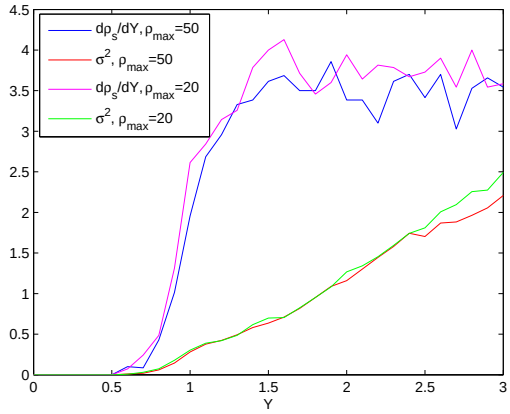

Figure 33: Comparison of velocity and variance between $\rho_{max} = 20$ and $\rho_{max} = 50$.

8 Appendix

8.1 Dependence of the model on ρ_{max}

During the final defense of this manuscript, the question was raised whether the length cutoff r_{min}, which in logarithmic coordinates is $\rho_{max} := \log_B \frac{1}{r_{min}}$, in the divergent integral (134) has any effect on the results of the model. Analytically, we can see from (67) that the BFKL equation in Mellin space does not have a cutoff dependence. In fact, the lower size bound ρ cancels in (62). Still, it may be asked whether this analytical cancellation applies to the model. I will demonstrate in several ways that the model does not have a strong dependence on ρ_{max} as long as it is sufficiently large.

8.1.1 Brute force model check

Running the model with different values of ρ_{max} is one way of checking for a possible dependence. For technical reasons explained before, it is not convenient to have $\rho_{max} \gtrsim 50$, but we may check smaller values. Figure 33, for example, compares $\rho_{max} = 20$ and $\rho_{max} = 50$. Over 30 powers of the logarithmic base, the change in velocity and variance is small, although it appears the front velocity is slightly higher for the $\rho_{max} = 20$ case. This may be due to the change in relative splitting probabilities between near-size and far-size splittings. However, we believe that for sufficiently large ρ_{max} the artifact of higher front velocities disappears, as we now explain.

8.1.2 Lifetime dependence

Using the collinear branching probability approximation made in (169), we can write

$$\left.\frac{dP_{i\to j}}{dY}\right|_{j>i} \approx 2\left(1+B^{-2(j-i)}\right) \tag{183}$$

Making the fixed impact parameter approximation and multiplying by $\left(\frac{r_j}{r_i}\right)^2 = B^{-2(j-i)}$ as an estimate of the probability that the daughter j will be created near the probe location,

$$\left.\frac{dP_{i\to j}}{dY}\right|_{j>i} \approx 2\left(B^{-2(j-i)}+B^{-4(j-i)}\right)$$

Now finding the total probability of a size i dipole splitting collinearly,

$$\frac{dP_i}{dY} \approx \sum_{j=i+1}^{\rho_{max}-1} 2\left(B^{-2(j-i)}+B^{-4(j-i)}\right) \tag{184}$$

Observe this sum is convergent as $\rho_{max}\to\infty$. Also, because it converges quickly, the effective dipole splitting rates (and lifetimes) are not highly sensitive to the exact value chosen for ρ_{max}, as long as $\rho_{max}\gg\rho_s(Y)$. By "effective", we mean the splittings that will affect the amplitudes measured at a particular impact parameter, which we estimated by adjusting the splitting probability by $B^{-2(j-i)}$.

8.1.3 Analytical check of BK equation using model constructs

We can explicitly check the BK equation (95) within the model construct to verify insensitivity to ρ_{max}. To do so, we want to investigate the collinear part of the integral from limits 0 to r_{min}, which in logarithmic coordinates $\rho := \log_B \frac{1}{x}$ become, respectively, ∞ and ρ_{max}. Writing the BK equation using logarithmic coordinates at some impact parameter and using (183),

$$\partial_Y N_i(Y) = \sum_{j=\rho_{max}}^{\infty} 2\left(1+B^{-2(j-i)}\right)\left[N_{f(i,j)}(Y)+N_j(Y)-N_i(Y)+N_{f(i,j)}(Y)N_j(Y)\right] \tag{185}$$

where $i := \log_B \frac{1}{x_{01}}$, $j := \log_B \frac{1}{x_{12}}$, and

$$\begin{aligned} f(i,j) &:= \log_B \frac{1}{x_{02}} \\ &= \log_B \frac{1}{\sqrt{x_{01}^2 + x_{12}^2 - 2x_{01}x_{12}\cos\phi}} \\ &= \frac{1}{2}\log_B \frac{B^{2i}}{1 + B^{2(j-i)} - 2B^{(j-i)}\cos\phi} \\ &\approx \frac{1}{2}\log_B \left[B^{2i}\left(1 + B^{-2(j-i)}\right)\right] \end{aligned}$$

Assume that Y is small enough such that $\rho_{max} \gg \rho_s(Y)$. This is required for the validity of the model, as the wavefront must "fit" within the allotted logarithmic domain. Then $j \gg \rho_s(Y)$ and $N_j(Y) \approx 0$ far ahead of the saturation front. Also, because we are in the collinear region, $j \gg i$, assuming ρ_{max} is large enough that this is possible, and thus $f(i,j) \approx i$. Therefore, we see that with a sufficiently large ρ_{max}, the term in brackets in (185) is approximately 0. Further increasing ρ_{max} will have little effect on $\partial_Y N_i(Y)$.

8.2 2D Code

```
1  /*
2  2D Dipole Simulation
3  Author: Matt Haley
4  Versions:
5  2: uses red black tree removal
6  3: uses openmp
7  -
8  */
9
10 #include <iostream>
11 #include <fstream>
12 #include <sstream>
13 #include <cstdlib>
14 #include <ctime>
15 #include <cmath>
16 #include <vector>
17 #include <omp.h>
18 #include <stdio.h>
19 #include <stdlib.h>
20
21 using namespace std;
22
23 #include "datastructs/RedBlackTree4.h"
24 #include <codecogs/stats/dists/discrete/discrete/randomsample.h>
25
26 // Declare global variables
27 const double B=2;
28 const double delta=1;
29 const double pi=3.1415926535;
30 const double epsilon=pow(10.0,-14);
31
```

```
32  double r( const int & i )
33  {
34    return pow(B,-i*delta);
35  }
36
37  bool areSame(double a, double b)
38  {
39      return abs(a - b) < epsilon;
40  }
41
42  int main()
43  {
44    // Seed random generator and make first call (predictable)
45    srand((unsigned)time(0));
46    rand( );
47
48    // Declare Input Vars -- all will be shared among threads and so should be const
49    const double Y_max=3;
50    const double Y1=1,Y2=2,Y3=3; // output at these Y
51    const double delta_Y=.1;
52    const int numEvents = 700;
53    const double kappa_cutoff = pow(10.0,-1.0);
54    //kappa_cutoff = 0; // disable cutoff
55    const double b_probe=0;
56    const double b_probe2=0; // make b_probeN=b_probe for faster runs at central IP
57    const double b_probe3=0;
58    const double b_probe4=0;
59    const double b_probe5=0;
60    //const double b_probe2=pow(10.0,-6.0); // make b_probeN=b_probe for faster runs at central IP
61    //const double b_probe3=pow(10.0,-4.0);
62    //const double b_probe4=pow(10.0,-2.0);
63    //const double b_probe5=pow(10.0,-1.0);
64    const int N_sat=25;
65    const int N_initial=N_sat;
66    //const int N_max=4*double(N_sat)/pow(kappa_cutoff,2); // max number of dipoles of a given size
67    //cout << "N_max = " << N_max << endl;
68    // double alpha_s=1;
69    // N_sat=delta/alpha_s^2;
70    const double probFactor = 2*log(B)/(2*pi);
71    const int rho_min = 0;
72    const int rho_max = 40;
73    //const int rho_max = 50;
74    const int n_azimuth = 12;
75    const double dphi = 2*pi/n_azimuth;
76
77    // Declare Other Vars
78
79    // using array for probability matrix instead of vector for multidimensionality
80    // first entry of last dimension is sum over theta
81    double prob_itoj[rho_max+1][rho_max+1][n_azimuth+1] = {{{0}}};
82    // initialize random generator, generate one value (predictable)
83    RandGen gen;
84    gen.RandInt(10);
85
86    // determine discrete probability matrix for i->j
87    for( int i=0; i<=rho_max; i++ )
88    {
89      for( int j=0; j<=rho_max; j++ )
90      {
91        double kthTerm;
92        if( j<i ) {
93          for( int k=3; k<=n_azimuth-3; k++) { // k limits depend on n_azimuth--here 60<k<300 deg
94            kthTerm = probFactor*dphi/(1+pow(B,2*(i-j))-2*pow(B,i-j)*cos(k*dphi));
95            prob_itoj[i][j][0] += kthTerm; // k=0 is total angular prob i->j
96            prob_itoj[i][j][k+1] = kthTerm; // prob for the kth angular bin
97          }
98        }
99        else if( j==i ) {
100         for( int k=2; k<=n_azimuth-2; k++) { // k limits depend on n_azimuth--here 60<k<300 deg
101           kthTerm = probFactor*dphi/(1+pow(B,2*(i-j))-2*pow(B,i-j)*cos(k*dphi));
```

```
102             prob_itoj[i][j][0] += kthTerm; // k=0 is total angular prob i->j
103             prob_itoj[i][j][k+1] = kthTerm; // prob for the kth angular bin
104           }
105         }
106         else { // j>i
107           for( int k=0; k<=n_azimuth-1; k++) {
108             kthTerm = probFactor*dphi/(1+pow(B,2*(i-j))-2*pow(B,i-j)*cos(k*dphi));
109             prob_itoj[i][j][0] += kthTerm; // k=0 is total angular prob i->j
110             prob_itoj[i][j][k+1] = kthTerm; // prob for the kth angular bin
111           }
112         }
113       }
114     }
115
116     // print probability matrix
117     cout << "Probability Matrix:" << endl;
118     cout << endl;
119     for( int i=0; i<=rho_max; i++)
120     {
121       for( int j=0; j<=rho_max; j++ ) {
122         cout << prob_itoj[i][j][0] << " ";
123       }
124       cout << endl;
125     }
126     cout << endl;
127
128     // determine lifetimes
129     vector<double> sum(rho_max+1);
130     vector<double> lifetime(rho_max+1); // upper limit on rho_a is rho_max-1??
131     for( int i=0; i<=rho_max; i++ ) {
132       sum[i] = 0;
133       for( int j=0; j<=rho_max; j++ ) {
134         sum[i] += prob_itoj[i][j][0];
135       }
136       lifetime[i] = 1/sum[i];
137     }
138     // output lifetimes
139     cout << "Lifetimes:" << endl;
140     for( int rho_a=rho_min; rho_a<=rho_max; rho_a++) {
141       cout << 1/lifetime[rho_a] << endl;
142     }
143
144     // print probability matrix, fixed i and j, print angular probabilities
145     /*
146     cout << endl;
147     int i2 = 0;
148     int j2 = 0;
149     cout << prob_itoj[i2][j2][0] << ": ";
150     for( int k=0; k<=n_azimuth-1; k++) {
151       cout << prob_itoj[i2][j2][k+1] << " ";
152     }
153     cout << endl;
154     i2 = 0;
155     j2 = 1;
156     cout << prob_itoj[i2][j2][0] << ": ";
157     for( int k=0; k<n_azimuth; k++) {
158       cout << prob_itoj[i2][j2][k+1] << " ";
159     }
160     cout << endl << endl;
161     */
162
163     vector< Stats::Dists::Discrete::Discrete::RandomSample<double>* > prob_itoj_gen(rho_max);
164     vector< Stats::Dists::Discrete::Discrete::RandomSample<double>* > prob_k_azimuth_gen(2*(rho_max
          -1)+1);
165     double passToGenij[rho_max][rho_max] = {{0}};
166     double passToGenk[2*(rho_max-1)+1][n_azimuth] = {{0}};
167     for(int i=0; i<=rho_max-1; i++) {
168       for(int j=0; j<=rho_max-1; j++) {
169         passToGenij[i][j] = prob_itoj[i][j][0];
170       }
```

```
171     }
172     for(int i=0; i<=rho_max-1; i++) {
173       prob_itoj_gen[i]=new Stats::Dists::Discrete::Discrete::RandomSample<double>(rho_max,
              passToGenij[i], true, time(0)/MERSENNEDIV);
174       //prob_itoj_gen[i]=new Stats::Dists::Discrete::Discrete::RandomSample<double>(rho_max,
              prob_itoj[i], true, 0.3416);
175     }
176     for(int j=0; j<=2*(rho_max-1); j++) {
177       if( j<=rho_max-2 ) {
178         for(int k=0; k<=n_azimuth-1; k++) {
179           passToGenk[j][k] = prob_itoj[rho_max-1][j][k+1]; // last row of prob matrix
180         }
181       }
182       else { // j>rho_max-2, where the rho_max-1 entry is for i->i
183         for(int k=0; k<=n_azimuth-1; k++) {
184           passToGenk[j][k] = prob_itoj[0][j][k+1]; // first row of prob matrix
185         }
186       }
187     }
188     for(int j=0; j<=2*(rho_max-1); j++) {
189       prob_k_azimuth_gen[j]=new Stats::Dists::Discrete::Discrete::RandomSample<double>(n_azimuth,
              passToGenk[j], true, time(0)/MERSENNEDIV);
190     }
191
192     cout << "Y_max = " << Y_max << ", kappa_cutoff = " << kappa_cutoff <<
193         ", events = " << numEvents << endl;
194     cout << "probFactor = " << probFactor << endl;
195
196     // end serial code initializers
197
198     #pragma omp parallel // clear contents of output files
199     {
200       int th_id = omp_get_thread_num();
201       ofstream fileOutputStream,rho_sStream,TatProbeY1,TatProbeY2,TatProbeY3;
202       stringstream ss;
203       ss << th_id;
204       string filename;
205
206       filename = "rho_sCore" + ss.str() + ".dat";
207       rho_sStream.open(filename.c_str()); // clears file contents
208       ////rho_sStream << "numEvents= " << numEvents << endl;
209       rho_sStream.close();
210
211       filename = "TatProbeY1Core" + ss.str() + ".dat";
212       TatProbeY1.open(filename.c_str()); // clears file contents
213       ////TatProbeY1 << "numEvents= " << numEvents << endl;
214       TatProbeY1.close();
215
216       filename = "TatProbeY2Core" + ss.str() + ".dat";
217       TatProbeY2.open(filename.c_str()); // clears file contents
218       ////TatProbeY2 << "numEvents= " << numEvents << endl;
219       TatProbeY2.close();
220
221       filename = "TatProbeY3Core" + ss.str() + ".dat";
222       TatProbeY3.open(filename.c_str()); // clears file contents
223       ////TatProbeY3 << "numEvents= " << numEvents << endl;
224       TatProbeY3.close();
225     }
226
227     // EVENT LOOP
228     cout << "Num procs = " << omp_get_num_procs() << endl;
229     #pragma omp parallel for
230     for( int event=1; event<=numEvents; event++ ) {
231
232     // initialize thread variables
233     double kappa,kappa2,kappa3,kappa4,kappa5;
234     double b01,b01x,b01y,x0x,x0y,x1x,x1y,x2x,x2y;
235     double x01x,x01y,x02x,x02y,x12x,x12y;
236     double length_x01,x01hatx,x01haty,length_x02,length_x12;
237     double b02,b12,b02x,b02y,b12x,b12y;
```

```
238     double b02hatx,b02haty,b12hatx,b12haty,checkpointx,checkpointy;
239     int rho_x02, rho_x12;
240     double numsplits_i;
241     bool sizeRangex02,sizeRangex12,exceedkappax02,exceedkappax12,unSatx02,unSatx02Lower,
            unSatx02Upper,
242       unSatx12,unSatx12Lower,unSatx12Upper;
243     double T[rho_max+1][5] = {{0}};
244     double b;
245     int rho_s,rho_sPrev;
246     double angle,angle1,angle2;
247     int count1,count2,count3,count4,count5,count6;
248     count1=count2=count3=count4=count5=count6=0;
249
250     // stream vars
251     int th_id = omp_get_thread_num();
252     ofstream fileOutputStream,rho_sStream,TatProbeY1,TatProbeY2,TatProbeY3;
253     streambuf* sbuf = cout.rdbuf(); // make a copy of the cout stream buffer
254     stringstream ss;
255     ss << th_id;
256     string filename;
257
258     filename = "rho_sCore" + ss.str() + ".dat";
259     rho_sStream.open(filename.c_str(),ios::app); // appends to file contents
260
261     filename = "TatProbeY1Core" + ss.str() + ".dat";
262     TatProbeY1.open(filename.c_str(),ios::app); // appends to file contents
263
264     filename = "TatProbeY2Core" + ss.str() + ".dat";
265     TatProbeY2.open(filename.c_str(),ios::app); // appends to file contents
266
267     filename = "TatProbeY3Core" + ss.str() + ".dat";
268     TatProbeY3.open(filename.c_str(),ios::app); // appends to file contents
269
270     cout << "////////////////////// EVENT = " << event << ", core = " <<
271         omp_get_thread_num() << " //////////////////////" << endl << endl;
272     // initialize n[i]
273     vector< RedBlackTree<double>* > n(rho_max+1);
274     for( int i=0; i<=rho_max; i++ ) {
275       n[i]=new RedBlackTree<double>(-1000);
276     }
277     // populate the initial size dipoles
278     for( int k_b=1; k_b<=N_initial; k_b++ )
279     {
280       angle1 = 2*pi*gen.RandReal();
281       angle2 = 2*pi*gen.RandReal();
282       b01 = r(0)/2*gen.RandReal();
283       b01x = b01*cos(angle1);
284       b01y = b01*sin(angle1);
285       x0x = b01x + r(0)/2*cos(angle2);
286       x0y = b01y + r(0)/2*sin(angle2);
287       x1x = b01x - r(0)/2*cos(angle2);
288       x1y = b01y - r(0)/2*sin(angle2);
289       n[0]->insert(b01,b01x,b01y,x0x,x0y,x1x,x1y);
290     }
291     rho_s=rho_sPrev=0;
292     // Rapidity Loop
293     for( double Y=0; Y<=Y_max+epsilon; Y=Y+delta_Y ) {
294       cout << "/////////////// Y = " << Y << endl << endl; // output progress
295       for( int i=0; i<=rho_max-1; i++ ) {
296         numsplits_i=1/lifetime[i]*delta_Y*n[i]->size();
297         if( numsplits_i != 0 ) {
298           //cout << "i=" << i << ", numsplits_i= " << numsplits_i << endl;
299           //cout << "lifetime=" << lifetime[i] << ", size=" << n[i]->size() << endl;
300           //cout << endl;
301         }
302         for( int l=1; l<=numsplits_i; l++ ) {
303           if( n[i]->size() > 0 ) { // only split if dipoles exist
304           // choose a random dipole from column i to split
305           n[i]->randElement(b01,b01x,b01y,x0x,x0y,x1x,x1y);
306           // choose size j to split into
```

```
307           int j=int( prob_itoj_gen[i]->genReal() );
308           // choose azimuth k to split into
309           int k = int( prob_k_azimuth_gen[j-i+(rho_max-1)]->genReal() );
310           // calculate x2
311           angle = 2*pi*double(k)/double(n_azimuth);
312           x01x = x1x - x0x;
313           x01y = x1y - x0y;
314           length_x01 = pow(pow(x01x,2)+pow(x01y,2),.5); //Pythagorean thm
315           x01hatx = x01x/length_x01;
316           x01haty = x01y/length_x01;
317           x2x = -r(j)*(cos(angle)*x01hatx - sin(angle)*x01haty); // just rotation piece
318           x2y = -r(j)*(sin(angle)*x01hatx + cos(angle)*x01haty); // just rotation piece
319           // choose which side of x01 to split off of
320           if(gen.RandInt(0,1)==1) { // split off of x1
321             x2x = x2x + x1x;
322             x2y = x2y + x1y;
323           }
324           else { // split off of x0
325             x2x = -x2x + x0x;
326             x2y = -x2y + x0y;
327           }
328           // choose IP to split into
329           b02x = (x0x + x2x)/2.0;
330           b02y = (x0y + x2y)/2.0;
331           b02 = pow(pow(b02x,2)+pow(b02y,2),.5);
332           b12x = (x1x + x2x)/2.0;
333           b12y = (x1y + x2y)/2.0;
334           b12 = pow(pow(b12x,2)+pow(b12y,2),.5);
335
336           x02x = x2x - x0x;
337           x02y = x2y - x0y;
338           length_x02 = pow(pow(x02x,2)+pow(x02y,2),.5); //Pythagorean thm
339           x12x = x2x - x1x;
340           x12y = x2y - x1y;
341           length_x12 = pow(pow(x12x,2)+pow(x12y,2),.5); //Pythagorean thm
342
343           // insert new dipoles, round new dipoles to nearest log_2
344           ////rho_x02 = floor(log(1/length_x02)/log(2) + .5);
345           ////rho_x12 = floor(log(1/length_x12)/log(2) + .5);
346           rho_x02 = int(log(1/length_x02)/log(B) + .5);
347           rho_x12 = int(log(1/length_x12)/log(B) + .5);
348
349           /*
350           cout << "x0={" << x0x << "," << x0y << "}, x1={" << x1x << "," << x1y <<
351                "}, x2={" << x2x << "," << x2y << "}" << endl;
352           cout << "   b01={" << b01x << "," << b01y << "}, b02={" << b02x <<
353                "," << b02y << "}, b12={" << b12x << "," << b12y << "}" << endl;
354           */
355           /*
356           cout << "endpoint x coordinates: " << x0x << "," << x1x << "," << x2x << endl;
357           cout << "endpoint y coordinates: " << x0y << "," << x1y << "," << x2y << endl;
358           cout << "IP x coordinates: " << b01x << "," << b02x << "," << b12x << endl;
359           cout << "IP y coordinates: " << b01y << "," << b02y << "," << b12y << endl;
360           cout << "rho_x01=" << i << ", rho_x02=" << rho_x02 << ", rho_x12=" << rho_x12 << endl;
361           cout << endl;
362           */
363
364           // check various conditions before adding daughters x02 and x12
365           unSatx02=unSatx02Lower=unSatx02Upper=unSatx12=unSatx12Lower=unSatx12Upper=0;
366           sizeRangex02 = (rho_x02 >= 0) && (rho_x02 <= rho_max);
367           sizeRangex12 = (rho_x12 >= 0) && (rho_x12 <= rho_max);
368
369           kappa = r(rho_x02)/abs(b02-b_probe);
370           kappa2 = r(rho_x02)/abs(b02-b_probe2);
371           kappa3 = r(rho_x02)/abs(b02-b_probe3);
372           kappa4 = r(rho_x02)/abs(b02-b_probe4);
373           kappa5 = r(rho_x02)/abs(b02-b_probe5);
374           exceedkappax02 = (kappa > kappa_cutoff) || (kappa2 > kappa_cutoff) || (kappa3 >
                   kappa_cutoff)
375                || (kappa4 > kappa_cutoff)  || (kappa5 > kappa_cutoff);
```

```
376           kappa = r(rho_x12)/abs(b12-b_probe);
377           kappa2 = r(rho_x12)/abs(b12-b_probe2);
378           kappa3 = r(rho_x12)/abs(b02-b_probe3);
379           kappa4 = r(rho_x12)/abs(b02-b_probe4);
380           kappa5 = r(rho_x12)/abs(b02-b_probe5);
381           exceedkappax12 = (kappa > kappa_cutoff) || (kappa2 > kappa_cutoff)|| (kappa3 >
                  kappa_cutoff)
382               || (kappa4 > kappa_cutoff)  || (kappa5 > kappa_cutoff);
383
384           if( sizeRangex02 && sizeRangex12 && exceedkappax02 && exceedkappax12 ) {
385             // assume already saturated if size > N_max
386             //if( ((n[rho_x02]->size() >= N_max) || (n[rho_x12]->size() >= N_max)) ) continue;
387             // if other tests passed, do time consuming saturation tests in nested form (nesting
                    saves computation)
388             unSatx02 = ( n[rho_x02]->between2D(b02-r(rho_x02)/2,b02+r(rho_x02)/2,r(rho_x02),b02x,
                    b02y) < N_sat );
389             if( unSatx02 ) { // lower boundary x02
390               b02hatx = b02x/b02;
391               b02haty = b02y/b02;
392               checkpointx = b02x-r(rho_x02)/2*b02hatx;
393               checkpointy = b02y-r(rho_x02)/2*b02haty;
394               if( b02-r(rho_x02)/2 >= 0 ) {
395                 unSatx02Lower = n[rho_x02]->between2D(b02-r(rho_x02),b02,r(rho_x02),checkpointx,
                        checkpointy) < N_sat;
396               }
397               else { // b02-r(rho_x02)/2 < 0
398                 unSatx02Lower = n[rho_x02]->between2D(0,abs(b02-r(rho_x02)),r(rho_x02),checkpointx,
                        checkpointy) < N_sat;
399               }
400               if ( unSatx02Lower ) { // upper boundary x02
401                 checkpointx = b02x+r(rho_x02)/2*b02hatx;
402                 checkpointy = b02y+r(rho_x02)/2*b02haty;
403                 unSatx02Upper = n[rho_x02]->between2D(b02,b02+r(rho_x02),r(rho_x02),checkpointx,
                        checkpointy) < N_sat;
404                 if( unSatx02Upper ) { // at b12
405                   unSatx12 = n[rho_x12]->between2D(b12-r(rho_x12)/2,b12+r(rho_x12)/2,r(rho_x12),b12x
                          ,b12y) < N_sat;
406                   if( unSatx12 ) { // lower boundary x02
407                     b12hatx = b12x/b12;
408                     b12haty = b12y/b12;
409                     checkpointx = b12x-r(rho_x12)/2*b12hatx;
410                     checkpointy = b12y-r(rho_x12)/2*b12haty;
411                     if( b12-r(rho_x12)/2 >= 0 ) {
412                       unSatx12Lower = n[rho_x12]->between2D(b12-r(rho_x12),b12,r(rho_x12),
                              checkpointx,checkpointy) < N_sat;
413                     }
414                     else { // b12-r(rho_x12)/2 < 0
415                       unSatx12Lower = n[rho_x12]->between2D(0,abs(b12-r(rho_x12)),r(rho_x12),
                              checkpointx,checkpointy) < N_sat;
416                     }
417                     if ( unSatx12Lower ) { // upper boundary x12
418                       checkpointx = b12x+r(rho_x12)/2*b12hatx;
419                       checkpointy = b12y+r(rho_x12)/2*b12haty;
420                       unSatx12Upper = n[rho_x12]->between2D(b12,b12+r(rho_x12),r(rho_x12),
                              checkpointx,checkpointy) < N_sat;
421                     } // upper boundary x12
422                   } // lower boundary x12
423                 } // at b12
424               } // upper boundary x02
425             } // lower boundary x02
426           } // sizerange and kappa check
427           //else {
428           //  cout << "veto sizerange or kappa: " << sizeRangex02 << ", " << sizeRangex12 << ", " <<
429           //      exceedkappax02 << ", " << exceedkappax12 << endl;
430           //}
431           if( unSatx02 && unSatx02Lower && unSatx02Upper && unSatx12 && unSatx12Lower &&
                  unSatx12Upper ) {
432             if( floor(b01*10000) != floor(pow( pow(b01x,2)+pow(b01y,2),.5 )*10000) ) {
433               cout << "*******ALERT*******: " << b01 << " " << pow( pow(b01x,2)+pow(b01y,2) ,.5 ) <<
                      endl << endl << endl << endl;
```

```
434             }
435             ///n[i]->printTreeVector();
436             n[i]->remove(b01); // remove parent
437             n[rho_x02]->insert(b02,b02x,b02y,x0x,x0y,x2x,x2y); // add b02
438             n[rho_x12]->insert(b12,b12x,b12y,x1x,x1y,x2x,x2y); // add b12
439             //cout << "x0x = " << x0x << ", x1x = " << x1x << ", x2x = " << x2x << endl;
440             //cout << "x0y = " << x0y << ", x1y = " << x1y << ", x2y = " << x2y << endl << endl;
441             //cout << "entry rho_x02=" << rho_x02 << ": " << b02 << "," << b02x << "," << b02y <<
                    "," << x0x << "," << x0y
442             //    << "," << x2x << "," << x2y << endl;
443             //cout << "entry rho_x12=" << rho_x12 << ": " << b12 << "," << b12x << "," << b12y <<
                    "," << x1x << "," << x1y
444             //    << "," << x2x << "," << x2y << endl;
445             //cout << "removal i=" << i << ": " << b01 << "," << b01x << "," << b01y << "," << x0x
                    << "," << x0y
446             //    << "," << x1x << "," << x1y << endl << endl;
447           } // saturation check
448           //else {
449           //  cout << "veto saturation: " << unSatx02 << ", " << unSatx02Lower << ", " <<
                  unSatx02Upper <<
450           //    ", " << unSatx12 << ", " << unSatx12Lower << ", " << unSatx12Upper <<endl;
451           //}
452         } // tree size check
453         } // dipole creation (numsplits)
454       } // i loop
455
456       // output amplitude
457       for(int i=0; i<=rho_max; i++) {
458         for(int j=0; j<=4; j++) {
459           if( j==0 ) b=0;
460           else if( j==1 ) b=pow(10.0,-6);
461           else if( j==2 ) b=pow(10.0,-4);
462           else if( j==3 ) b=pow(10.0,-2);
463           else b=pow(10.0,-1); // j==4
464           T[i][j] = (double) n[i]->between2D(b-r(i)/2,b+r(i)/2,r(i),b,0)/( (double) N_sat);
465         }
466       }
467       if( areSame(Y,Y1) || areSame(Y,Y2) || areSame(Y,Y3) ) {
468         cout << "OUTPUTTING AMPLITUDE" << endl << endl;
469         if( areSame(Y,Y1) ) {
470           for(int i=0; i<=rho_max; i++) {
471             TatProbeY1 << i << " " << T[i][0] << endl;
472           }
473           TatProbeY1 << "end_of_event=" << event << endl;
474         }
475         if( areSame(Y,Y2) ) {
476           for(int i=0; i<=rho_max; i++) {
477             TatProbeY2 << i << " " << T[i][0] << endl;
478           }
479           TatProbeY2 << "end_of_event=" << event << endl;
480         }
481         if( areSame(Y,Y3) ) {
482           for(int i=0; i<=rho_max; i++) {
483             TatProbeY3 << i << " " << T[i][0] << endl;
484           }
485           TatProbeY3 << "end_of_event=" << event << endl;
486         }
487         /*for(int i=0; i<=rho_max-1; i++) {
488           fileOutputStream << T[i] << endl;
489         }*/
490         ///fileOutputStream.close();
491         //cout.rdbuf(sbuf); // reassign cout to console output
492       } // end output
493
494       // front position at central IP
495       rho_sStream << Y << " ";
496       for( int j=0; j<=4; j++ ) {
497         for(int i=0; i<=rho_max; i++) {
498           if( T[i][j] >= .5 ) rho_s=i;
499         }
```

```
500         rho_sStream << rho_s << " ";
501       }
502       rho_sStream << endl;
503     } // Y loop
504
505     fileOutputStream.open("TAmplitude.dat");
506     fileOutputStream << "b,b01x,b01y,x0x,x0y,x1x,x1y,kappa=" << kappa_cutoff << ",Y_max=" << Y_max
          <<
507             ",numEvents=" << numEvents << ",probFactor=" << probFactor << endl;
508     cout.rdbuf(fileOutputStream.rdbuf()); // redirect cout to the output file stream
509     for( int i=0; i<=rho_max; i++ ) {
510       ///cout << "i=" << i << endl;
511       ///n[i]->printTreeVector();
512     }
513     cout.rdbuf(sbuf); // reassign cout to console output
514     fileOutputStream.close();
515     rho_sStream << "end_of_event=" << event << endl;
516     rho_sStream.close();
517     TatProbeY1.close();
518     TatProbeY2.close();
519     TatProbeY3.close();
520     } // EVENT LOOP, threads rejoin
521
522     // compile data
523     int numThreads = omp_get_num_procs();
524     ofstream fileOutputStream,rho_sStream,TatProbeY1,TatProbeY2,TatProbeY3;
525     ifstream input;
526     stringstream ss;
527     rho_sStream.open("rho_s.dat");
528     rho_sStream << "numEvents= " << numEvents << endl;
529     TatProbeY1.open("TatProbeY1.dat");
530     TatProbeY2.open("TatProbeY2.dat");
531     TatProbeY3.open("TatProbeY3.dat");
532     TatProbeY1 << "numEvents= " << numEvents << endl;
533     TatProbeY2 << "numEvents= " << numEvents << endl;
534     TatProbeY3 << "numEvents= " << numEvents << endl;
535     for( int i=0; i<numThreads; i++ ) {
536       ss.str(""); // empty the string
537       ss << i;
538       string filename,data;
539
540       filename = "rho_sCore" + ss.str() + ".dat";
541       input.open(filename.c_str());
542       if( !input.fail() ) {
543         while( getline(input,data) ) {
544           rho_sStream << data << endl;
545         }
546       }
547       else cout << "Error: cannot open file " << filename << endl;
548       input.close();
549
550       filename = "TatProbeY1Core" + ss.str() + ".dat";
551       input.open(filename.c_str());
552       if( !input.fail() ) {
553         while( getline(input,data) ) {
554           TatProbeY1 << data << endl;
555         }
556       }
557       else cout << "Error: cannot open file" << filename << endl;
558       input.close();
559
560       filename = "TatProbeY2Core" + ss.str() + ".dat";
561       input.open(filename.c_str());
562       if( !input.fail() ) {
563         while( getline(input,data) ) {
564           TatProbeY2 << data << endl;
565         }
566       }
567       else cout << "Error: cannot open file" << filename << endl;
568       input.close();
```

```
569
570       filename = "TatProbeY3Core" + ss.str() + ".dat";
571       input.open(filename.c_str());
572       if( !input.fail() ) {
573         while( getline(input,data) ) {
574           TatProbeY3 << data << endl;
575         }
576       }
577       else cout << "Error: cannot open file" << filename << endl;
578       input.close();
579     }
580     rho_sStream.close();
581     TatProbeY1.close();
582     TatProbeY2.close();
583     TatProbeY3.close();
584
585     cout << "DONE, Y_max = " << Y_max << ", kappa_cutoff = " << kappa_cutoff <<
586         ", events = " << numEvents << endl;
587     cout << "probFactor = " << probFactor << endl;
588     ///cout << "saturation veto counts: " << count1 << ", " << count2 << ", " << count3 << ", " <<
589     ///   count4 << ", " << count5 << ", " << count6 << endl;
590     cout << flush;
591     return 0;
592   }
```

8.3 2DR Code Snippet

```
1  // 2DR changes
2  x1xPrime = b12x + length_x12/2*x01hatx;
3  x2xPrime = b12x - length_x12/2*x01hatx;
4  x0xPrimePrime = b02x - length_x02/2*x01hatx;
5  x2xPrimePrime = b02x + length_x02/2*x01hatx;
6  b02y = 0;
7  b12y = 0;
8  b02 = abs(b02x);
9  b12 = abs(b12x);
```

8.4 2DSR Code Snippet

```
1  // 2DSR changes
2  s = gen.RandReal(0,1);
3  smallerRho = min(rho_x02,rho_x12);
4  stripwidth = stripFactor*r(smallerRho);
5  if( abs(x2y) > stripwidth ) {
6    if( x2y < 0 ) s = -s; // project to the correct side of the x-axis
7    x2xPrime = x2x;
8    x2yPrime = s*stripwidth;
9    if( length_x02 > abs(x0y-x2yPrime) ) {
10     if( x0x > x2xPrime ) { // x0 slides up x-axis
11       x0xPrime = x2xPrime + pow(pow(length_x02,2)-pow(x0y-x2yPrime,2),.5);
12     }
13     else { // x0x <= x2xPrime, x0 slides down x-axis
14       x0xPrime = x2xPrime - pow(pow(length_x02,2)-pow(x0y-x2yPrime,2),.5);
15     }
16   }
17   else x0xPrime = x0x;
18   if( length_x12 > abs(x1y-x2yPrime) ) {
19     if( x1x > x2xPrime ) { // x1 slides up x-axis
20       x1xPrime = x2xPrime + pow(pow(length_x12,2)-pow(x1y-x2yPrime,2),.5);
21     }
22     else { // x1x <= x2xPrime, x1 slides down x-axis
23       x1xPrime = x2xPrime - pow(pow(length_x12,2)-pow(x1y-x2yPrime,2),.5);
24     }
25   }
26   else x1xPrime = x1x;
27   if( x0xPrime!=x0xPrime || x1xPrime!=x1xPrime ) {
```

```
28        cout << "*******ALERT*******" << endl;
29      }
30      x0yPrime = x0y;
31      x1yPrime = x1y;
32
33      // redefine impact parameters to primed ones
34      b02x = (x0xPrime+x2xPrime)/2.0;
35      b02y = (x0yPrime+x2yPrime)/2.0;
36      b02 = pow(pow(b02x,2)+pow(b02y,2),.5);
37      b12x = (x1xPrime+x2xPrime)/2.0;
38      b12y = (x1yPrime+x2yPrime)/2.0;
39      b12 = pow(pow(b12x,2)+pow(b12y,2),.5);
40      // redefine coordinates to primed ones
41      x0x = x0xPrime;
42      x0y = x0yPrime;
43      x1x = x1xPrime;
44      x1y = x1yPrime;
45      x2x = x2xPrime;
46      x2y = x2yPrime;
47    }
```

8.5 RedBlackTree.h

```
1           #ifndef RED_BLACK_TREE_H_
2           #define RED_BLACK_TREE_H_
3
4           #include "datastructs/dsexceptions.h"
5           #include <iostream>        // For NULL
6       #include <cstdlib>
7       #include "tapestry/randgen.h"
8
9           // Red-black tree class
10          //
11          // CONSTRUCTION: with negative infinity object also
12          //                used to signal failed finds
13          //
14          // ******************PUBLIC OPERATIONS*********************
15          // void insert( x )       --> Insert x
16          // void remove( x )       --> Remove x (unimplemented)
17          // Comparable find( x )   --> Return item that matches x
18          // Comparable findMin( )  --> Return smallest item
19          // Comparable findMax( )  --> Return largest item
20          // boolean isEmpty( )     --> Return true if empty; else false
21          // void makeEmpty( )      --> Remove all items
22          // void printTree( )      --> Print tree in sorted order
23      // void printTree2( )   --> Print tree in tree order
24      // int size( )         --> Returns number of nodes in tree
25      // int between(x, y)    --> Returns number of nodes with elements between x and y
26      // Comparable randElement(n) --> Returns a random tree element from the first n nodes
27
28            // Node and forward declaration because g++ does
29            // not understand nested classes.
30          template <class Comparable>
31          class RedBlackTree;
32
33          template <class Comparable>
34          class RedBlackNode
35          {
36              Comparable    element;
37              ///RedBlackNode *left;
38              ///RedBlackNode *right;
39              RedBlackNode *link[2]; // Left (0) and right (1) links
40              int           red;
41              double       bx;
42              double       by;
43              double       x0x;
44              double       x0y;
45              double       x1x;
```

```
46              double       x1y;
47
48              // c = 0 should be c = RedBlackTree<Comparable>::BLACK
49              // But Visual 5.0 does not comprehend it.
50              RedBlackNode( const Comparable & theElement = Comparable( ),
51                            RedBlackNode *lt = NULL, RedBlackNode *rt = NULL,
52                            double thebx = double(), double theby = double(),
53                            double thex0x = double(), double thex0y = double(),
54                            double thex1x = double(), double thex1y = double(),
55                            int thered = 1 )
56                : element( theElement ), red( thered ), bx(thebx),
57                  by(theby), x0x(thex0x), x0y(thex0y), x1x(thex1x), x1y(thex1y)
58                  {
59                  link[0] = lt;
60                  link[1] = rt;
61                  ///cout << "element= " << element << ", mem of link[1]=" << link[1] << endl;
62                  }
63              friend class RedBlackTree<Comparable>;
64          };
65
66          template <class Comparable>
67          class RedBlackTree
68          {
69            public:
70              explicit RedBlackTree( const Comparable & negInf );
71              RedBlackTree( const RedBlackTree & rhs );
72              ~RedBlackTree( );
73
74              const Comparable & findMin( ) const;
75              const Comparable & findMax( ) const;
76              const Comparable & find( const Comparable & x ) const;
77              bool isEmpty( ) const;
78              void printTree( ) const;
79        void printTree2( ) const;
80        void printTreeVector( ) const;
81        int size( ) const;
82        int between(const Comparable & lower, const Comparable & upper) const;
83        int between2D(const Comparable & lower, const Comparable & upper,
84            const double & ri, const double & thebx, const double & theby) const;
85  //      void randElement( const int & n, Comparable & theElement,
86  //                  double & thebx, double & theby,
87  //                  double & thex0x, double & thex0y,
88  //                  double & thex1x, double & thex1y) const;
89        void randElement( Comparable & theElement,
90            double & thebx, double & theby,
91            double & thex0x, double & thex0y,
92            double & thex1x, double & thex1y) const;
93
94              void makeEmpty( );
95              //void insert( const Comparable & x );
96              int insert( const Comparable & x, const double & bx,
97                          const double & by, const double & x0x,
98                          const double & x0y, const double & x1x,
99                          const double & x1y);
100             int remove( const Comparable & x );
101
102             enum { BLACK, RED };
103
104             const RedBlackTree & operator=( const RedBlackTree & rhs );
105
106           private:
107             RedBlackNode<Comparable> *header;   // The tree header (contains negInf)
108             const Comparable ITEM_NOT_FOUND;
109             RedBlackNode<Comparable> *nullNode;
110
111                 // Used in insert routine and its helpers (logically static)
112             RedBlackNode<Comparable> *current;
113             RedBlackNode<Comparable> *parent;
114             RedBlackNode<Comparable> *grand;
115             RedBlackNode<Comparable> *great;
```

```
116
117                 // Usual recursive stuff
118             void reclaimMemory( RedBlackNode<Comparable> *t ) const;
119             void printTree( RedBlackNode<Comparable> *t ) const;
120         void printTree2( RedBlackNode<Comparable> *t ) const;
121         void printTreeVector( RedBlackNode<Comparable> *t ) const;
122         int recursiveSize( RedBlackNode<Comparable> *t ) const;
123         int between(const Comparable & lower, const Comparable & upper, RedBlackNode<Comparable> *t)
                const;
124         int between2D(const Comparable & lower, const Comparable & upper, RedBlackNode<Comparable> *
                t,
125             const double & ri, const double & thebx, const double & theby) const;
126               void randElement( RedBlackNode<Comparable> *t, int & countdown, Comparable &
                      theElement,
127                     double & thebx, double & theby,
128                     double & thex0x, double & thex0y,
129                     double & thex1x, double & thex1y, bool & done ) const;
130         RedBlackNode<Comparable> * clone( RedBlackNode<Comparable> * t ) const;
131
132                 // Red-black tree manipulations
133         RedBlackNode<Comparable> * jsw_single(RedBlackNode<Comparable> *root, int dir) const;
134         RedBlackNode<Comparable> * jsw_double(RedBlackNode<Comparable> *root, int dir) const;
135               //void handleReorient( const Comparable & item );
136               //RedBlackNode<Comparable> * rotate( const Comparable & item,
137               //                                   RedBlackNode<Comparable> *parent ) const;
138               //void rotateWithLeftChild( RedBlackNode<Comparable> * & k2 ) const;
139               //void rotateWithRightChild( RedBlackNode<Comparable> * & k1 ) const;
140
141               int is_red( RedBlackNode<Comparable> *root ) const;
142               int mySize;
143           };
144
145           #include "RedBlackTree4.cpp"
146           #endif
```

8.6 RedBlackTree.cpp

```
1  #include "RedBlackTree4.h"
2  #ifndef HEIGHT_LIMIT
3  #define HEIGHT_LIMIT 64 /* Tallest allowable tree */
4  #endif
5
6  /**
7   * Construct the tree.
8   * negInf is a value less than or equal to all others.
9   * It is also used as ITEM_NOT_FOUND.
10  */
11 template <class Comparable>
12 RedBlackTree<Comparable>::RedBlackTree( const Comparable & negInf )
13   : ITEM_NOT_FOUND( negInf )
14 {
15   nullNode     = new RedBlackNode<Comparable>;
16   nullNode->link[0] = nullNode->link[1] = nullNode;
17   header       = new RedBlackNode<Comparable>( negInf );
18   header->link[0] = header->link[1] = nullNode;
19   mySize = 0;
20 }
21
22 /**
23  * Copy constructor.
24  */
25 template <class Comparable>
26 RedBlackTree<Comparable>::RedBlackTree( const RedBlackTree<Comparable> & rhs )
27   : ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND ), mySize(rhs.mySize)
28 {
29   nullNode     = new RedBlackNode<Comparable>;
30   nullNode->link[0] = nullNode->link[1] = nullNode;
31   header       = new RedBlackNode<Comparable>( ITEM_NOT_FOUND );
```

```
32      header->link[0] = header->link[1] = nullNode;
33      *this = rhs;
34    }
35
36    /**
37     * Destroy the tree.
38     */
39    template <class Comparable>
40    RedBlackTree<Comparable>::~RedBlackTree( )
41    {
42      makeEmpty( );
43      delete nullNode;
44      delete header;
45    }
46
47
48    /**
49     * Remove item x from the tree.
50     * Not implemented in this version.
51     */
52    template <class Comparable>
53    int RedBlackTree<Comparable>::remove( const Comparable & x )
54    {
55      if ( header->link[1] != nullNode ) {
56        RedBlackNode<Comparable> head; /* False tree root */
57        RedBlackNode<Comparable> *q, *p, *g; /* Helpers */
58        RedBlackNode<Comparable> *f = nullNode;  /* Found item */
59          int dir = 1;
60
61          /* Set up our helpers */
62          q = &head;
63          g = p = nullNode;
64          q->link[0] = nullNode; // added so that looking above the root does not cause problems
65          q->link[1] = header->link[1];
66
67          /*
68            Search and push a red node down
69            to fix red violations as we go
70          */
71          while ( q->link[dir] != nullNode ) {
72            int last = dir;
73
74            /* Move the helpers down */
75            g = p;
76            p = q;
77            q = q->link[dir];
78            dir = q->element < x;
79
80            /*
81              Save the node with matching data and keep
82              going; we'll do removal tasks at the end
83            */
84            if ( q->element == x )
85              f = q;
86
87            /* Push the red node down with rotations and color flips */
88            if ( !is_red(q) && !is_red(q->link[dir]) ) {
89              if ( is_red ( q->link[!dir] ) )
90                p = p->link[last] = jsw_single ( q, dir );
91              else if ( !is_red ( q->link[!dir] ) ) {
92                RedBlackNode<Comparable> *s = p->link[!last];
93
94                if ( s != nullNode ) {
95                  if ( !is_red(s->link[!last]) && !is_red(s->link[last]) ) {
96                    /* Color flip */
97                    p->red = 0;
98                    s->red = 1;
99                    q->red = 1;
100                 }
101                 else {
```

```
102                   int dir2 = g->link[1] == p;
103
104                   if ( is_red ( s->link[last] ) )
105                     g->link[dir2] = jsw_double ( p, last );
106                   else if ( is_red ( s->link[!last] ) )
107                     g->link[dir2] = jsw_single ( p, last );
108
109                   /* Ensure correct coloring */
110                   q->red = g->link[dir2]->red = 1;
111                   g->link[dir2]->link[0]->red = 0;
112                   g->link[dir2]->link[1]->red = 0;
113                 }
114               }
115             }
116           }
117         } // end while
118
119         /* Replace and remove the saved node */
120         if ( f != nullNode ) {
121           ///tree->rel ( f->element );
122           f->element = q->element;
123           f->bx = q->bx;
124           f->by = q->by;
125           f->x0x = q->x0x;
126           f->x0y = q->x0y;
127           f->x1x = q->x1x;
128           f->x1y = q->x1y;
129           p->link[p->link[1] == q] =
130               q->link[q->link[0] == nullNode];
131           delete(q);
132           mySize--;
133         }
134
135         /* Update the root (it may be different) */
136         header->link[1] = head.link[1];
137
138         /* Make the root black for simplified logic */
139         if ( header->link[1] != nullNode )
140           header->link[1]->red = 0;
141
142         ///--tree->size;
143     }
144
145     return 1;
146   }
147
148   /**
149    * Find the smallest item  the tree.
150    * Return the smallest item or ITEM_NOT_FOUND if empty.
151    */
152   template <class Comparable>
153   const Comparable & RedBlackTree<Comparable>::findMin( ) const
154   {
155     if( isEmpty( ) )
156       return ITEM_NOT_FOUND;
157
158     RedBlackNode<Comparable> *itr = header->link[1];
159
160     while( itr->link[0] != nullNode )
161       itr = itr->link[0];
162
163     return itr->element;
164   }
165
166   /**
167    * Find the largest item in the tree.
168    * Return the largest item or ITEM_NOT_FOUND if empty.
169    */
170   template <class Comparable>
171   const Comparable & RedBlackTree<Comparable>::findMax( ) const
```

```
172   {
173     if( isEmpty( ) )
174       return ITEM_NOT_FOUND;
175
176     RedBlackNode<Comparable> *itr = header->link[1];
177
178     while( itr->link[1] != nullNode )
179       itr = itr->link[1];
180
181     return itr->element;
182   }
183
184   /**
185    * Find item x in the tree.
186    * Return the matching item or ITEM_NOT_FOUND if not found.
187    */
188   template <class Comparable>
189   const Comparable & RedBlackTree<Comparable>::find( const Comparable & x ) const
190   {
191     nullNode->element = x;
192     RedBlackNode<Comparable> *curr = header->link[1];
193
194     for( ; ; )
195     {
196       if( x < curr->element )
197         curr = curr->link[0];
198       else if( curr->element < x )
199         curr = curr->link[1];
200       else if( curr != nullNode )
201         return curr->element;
202       else
203         return ITEM_NOT_FOUND;
204     }
205   }
206
207   /**
208    * Make the tree logically empty.
209    */
210   template <class Comparable>
211   void RedBlackTree<Comparable>::makeEmpty( )
212   {
213     reclaimMemory( header->link[1] );
214     header->link[1] = nullNode;
215   }
216
217   /**
218    * Test if the tree is logically empty.
219    * Return true if empty, false otherwise.
220    */
221   template <class Comparable>
222   bool RedBlackTree<Comparable>::isEmpty( ) const
223   {
224     return header->link[1] == nullNode;
225   }
226
227   /**
228    * Print the tree contents in sorted order.
229    */
230   template <class Comparable>
231   void RedBlackTree<Comparable>::printTree( ) const
232   {
233     if( header->link[1] == nullNode )
234       cout << "Empty tree" << endl;
235     else
236       printTree( header->link[1] );
237   }
238
239   /**
240    * Print the tree contents in binary tree order.
241   */
```

```
242  template <class Comparable>
243  void RedBlackTree<Comparable>::printTree2( ) const
244  {
245    if( header->link[1] == nullNode )
246      cout << "Empty tree" << endl;
247    else
248      printTree2( header->link[1] );
249  }
250
251  /**
252   * Print the tree contents in order sorted.
253   */
254  template <class Comparable>
255  void RedBlackTree<Comparable>::printTreeVector( ) const
256  {
257    if( header->link[1] == nullNode )
258      cout << "Empty tree" << endl;
259    else
260      printTreeVector( header->link[1] );
261  }
262
263  /**
264   * Returns the number of nodes in the binary tree
265  */
266  template <class Comparable>
267  int RedBlackTree<Comparable>::size() const
268  {
269    return mySize;
270
271    //if( header->link[1] == nullNode )
272    //  return 0;
273    //else
274    //  return size( header->link[1] );
275  }
276
277  /**
278   * Returns the number of node elements between lower and upper
279   */
280  template <class Comparable>
281  int RedBlackTree<Comparable>::between(const Comparable & lower, const Comparable & upper) const
282  {
283    if( header->link[1] == nullNode)
284      return 0;
285    else
286      return between(lower,upper,header->link[1]);
287  }
288
289  /**
290  * Returns the number of ode elements between lower and upper and within a radius r_i
291  */
292  template <class Comparable>
293  int RedBlackTree<Comparable>::between2D(const Comparable & lower, const Comparable & upper,
294      const double & ri, const double & thebx, const double & theby) const
295  {
296    if( header->link[1] == nullNode)
297      return 0;
298    else
299      return between2D(lower,upper,header->link[1],ri,thebx,theby);
300  }
301
302  /**
303   * Returns a random element between the 1st and nth nodes (in order)
304   */
305  template <class Comparable>
306  void RedBlackTree<Comparable>::randElement( Comparable & theElement,
307      double & thebx, double & theby,
308      double & thex0x, double & thex0y,
309      double & thex1x, double & thex1y) const
310  {
311    if( header->link[1] == nullNode) {
```

```
312       cout << "error, randElement called on empty tree" << endl;
313       return;
314     }
315 
316     RandGen gen;     // random number generator
317     gen.RandInt( 1,mySize ); // first predictable
318     int random_integer = gen.RandInt( 1,mySize );
319     bool thedone = 0;
320     return randElement( header->link[1], random_integer, theElement, thebx, theby, thex0x, thex0y,
321         thex1x, thex1y, thedone );
322   }
323 
324   /**
325    * Deep copy.
326    */
327   template <class Comparable>
328   const RedBlackTree<Comparable> &
329   RedBlackTree<Comparable>::operator=( const RedBlackTree<Comparable> & rhs )
330   {
331     if( this != &rhs )
332     {
333       makeEmpty( );
334       header->link[1] = clone( rhs.header->link[1] );
335     }
336 
337     return *this;
338   }
339 
340   /**
341    * Internal method to print a subtree t in sorted order.
342    */
343   template <class Comparable>
344   void RedBlackTree<Comparable>::printTree( RedBlackNode<Comparable> *t ) const
345   {
346     if( t != t->link[0] )
347     {
348       printTree( t->link[0] );
349       cout << t->element << endl;
350       printTree( t->link[1] );
351     }
352   }
353 
354   /**
355    * Internal method to print a subtree t in binary tree order.
356   */
357   template <class Comparable>
358   void RedBlackTree<Comparable>::printTree2( RedBlackNode<Comparable> *t ) const
359   {
360     if( t != t->link[0] )
361     {
362       cout << t->element << endl;
363       printTree2( t->link[0] );
364       printTree2( t->link[1] );
365     }
366   }
367 
368   /**
369    * Internal method to print a subtree t in sorted order.
370    */
371   template <class Comparable>
372   void RedBlackTree<Comparable>::printTreeVector( RedBlackNode<Comparable> *t ) const
373   {
374     if( t != t->link[0] )
375     {
376       printTreeVector( t->link[0] );
377       cout << t->element << " " << t->bx << " " << t->by << " " <<
378           t->x0x << " " << t->x0y << " "<< t->x1x << " " << t->x1y << endl;
379       printTreeVector( t->link[1] );
380     }
381   }
```

```
382
383  /**
384   * Internal method to return the number of nodes in the binary tree
385  */
386  template <class Comparable>
387  int RedBlackTree<Comparable>::recursiveSize( RedBlackNode<Comparable> *t ) const
388  {
389    if (t == t->link[0] )
390      return 0;
391    else
392      return 1+size( t->link[0] )+size( t->link[1] );
393  }
394
395  /**
396   * Internal method to return the number of node elements between x and y
397   */
398  template <class Comparable>
399  int RedBlackTree<Comparable>::between(const Comparable & lower, const Comparable & upper,
       RedBlackNode<Comparable> *t) const
400  {
401    if( t == t->link[0] )
402      return 0;
403    else if( t->element > lower && t->element < upper)
404      return 1+between( lower,upper,t->link[0] )+between( lower,upper,t->link[1]);
405    else if( t->element > lower )
406      return between( lower,upper,t->link[0]);
407    else if( t->element < upper )
408      return between( lower,upper,t->link[1]);
409    else
410    {
411      cout << "error" << endl;
412      return 0;
413    }
414  }
415
416  /**
417   * Internal method to return the number of node elements between x and y and within radius r_i
418   */
419  template <class Comparable>
420  int RedBlackTree<Comparable>::between2D(const Comparable & lower, const Comparable & upper,
421      RedBlackNode<Comparable> *t, const double & ri, const double & thebx, const double & theby)
           const
422  {
423    if( t == t->link[0] )
424      return 0;
425    else if( t->element >= lower && t->element <= upper) {
426      ///if( t->hasSplit == 0 ) {
427      double dist = pow(pow(thebx - t->bx,2) + pow(theby - t->by,2) ,.5);
428      if( dist <= ri/2 ) { // check vector distance
429        return 1+between2D(lower,upper,t->link[0],ri,thebx,theby)+
430            between2D(lower,upper,t->link[1],ri,thebx,theby);
431      }
432      else {  // not within vector distance, keep looking
433        return 0+between2D(lower,upper,t->link[0],ri,thebx,theby)+
434            between2D(lower,upper,t->link[1],ri,thebx,theby);
435      }
436    }
437      ///else {// already split, don't count for saturation
438      /// return 0+between2D(lower,upper,t->link[0],ri,thebx,theby)+
439      ///      between2D(lower,upper,t->link[1],ri,thebx,theby);
440      ///}
441    else if( t->element > lower )
442      return between2D(lower,upper,t->link[0],ri,thebx,theby);
443    else if( t->element < upper )
444      return between2D(lower,upper,t->link[1],ri,thebx,theby);
445    else
446    {
447      cout << "error, between2D failed" << endl;
448      return 0;
449    }
```

```
450   }
451
452   /**
453    * Internal method to return the randomly chosen dipole
454   */
455   template <class Comparable>
456   void RedBlackTree<Comparable>::randElement( RedBlackNode<Comparable> *t, int & countdown,
457       Comparable & theElement,
458       double & thebx, double & theby,
459       double & thex0x, double & thex0y,
460       double & thex1x, double & thex1y, bool & thedone ) const
461   {
462     if( thedone == 1) return;
463     countdown--;
464     if( t == nullNode ) {
465       countdown++;
466       return;
467     }
468     else if( countdown == 0) {
469       theElement = t->element;
470       thebx = t->bx;
471       theby = t->by;
472       thex0x = t->x0x;
473       thex0y = t->x0y;
474       thex1x = t->x1x;
475       thex1y = t->x1y;
476       ///countdown = -1000;
477       thedone = 1;
478       return;
479     }
480     else {
481       randElement( t->link[0], countdown, theElement, thebx, theby, thex0x, thex0y, thex1x, thex1y,
            thedone );
482       randElement( t->link[1], countdown, theElement, thebx, theby, thex0x, thex0y, thex1x, thex1y,
            thedone );
483       return;
484     }
485   }
486
487   /**
488    * Internal method to clone subtree.
489    */
490   template <class Comparable>
491   RedBlackNode<Comparable> *
492   RedBlackTree<Comparable>::clone( RedBlackNode<Comparable> * t ) const
493   {
494     if( t == t->link[0] )  // Cannot test against nullNode!!!
495       return nullNode;
496     else
497       return new RedBlackNode<Comparable>( t->element, clone( t->link[0] ),
498                         clone( t->link[1] ), t->color, t->bx, t->by,
499                         t->x0x, t->x0y, t->x1x, t->x1y);
500   }
501
502
503   /**
504     <summary>
505     Performs a single red black rotation in the specified direction
506     This function assumes that all nodes are valid for a rotation
507     <summary>
508     <param name="root">The original root to rotate around</param>
509     <param name="dir">The direction to rotate (0 = left, 1 = right)</param>
510     <returns>The new root ater rotation</returns>
511     <remarks>For jsw_rbtree.c internal use only</remarks>
512   */
513   template <class Comparable>
514   RedBlackNode<Comparable> * RedBlackTree<Comparable>::jsw_single( RedBlackNode<Comparable> *root,
        int dir ) const
515   {
516     RedBlackNode<Comparable> *save = root->link[!dir];
```

```
517
518     root->link[!dir] = save->link[dir];
519     save->link[dir] = root;
520
521     root->red = 1;
522     save->red = 0;
523
524     return save;
525   }
526
527   /**
528     <summary>
529     Performs a double red black rotation in the specified direction
530     This function assumes that all nodes are valid for a rotation
531     <summary>
532     <param name="root">The original root to rotate around</param>
533     <param name="dir">The direction to rotate (0 = left, 1 = right)</param>
534     <returns>The new root after rotation</returns>
535     <remarks>For jsw_rbtree.c internal use only</remarks>
536   */
537   template <class Comparable>
538   RedBlackNode<Comparable> * RedBlackTree<Comparable>::jsw_double( RedBlackNode<Comparable> *root,
          int dir ) const
539   {
540     root->link[!dir] = jsw_single ( root->link[!dir], !dir );
541
542     return jsw_single ( root, dir );
543   }
544
545   /**
546     <summary>
547     Insert a copy of the user-specified
548     data into a red black tree
549     <summary>
550     <param name="tree">The tree to insert into</param>
551     <param name="data">The data value to insert</param>
552     <returns>
553     1 if the value was inserted successfully,
554     0 if the insertion failed for any reason
555     </returns>
556   */
557   template <class Comparable>
558   int RedBlackTree<Comparable>::insert( const Comparable & x, const double & bx,
559       const double & by, const double & x0x, const double & x0y, const double & x1x,
560       const double & x1y )
561   {
562     if( header->link[1] == nullNode) {
563       /*
564         We have an empty tree; attach the
565         new node directly to the root
566       */
567       header->link[1] = new RedBlackNode<Comparable>( x, nullNode, nullNode, bx, by, x0x, x0y, x1x,
              x1y );
568
569       if ( header->link[1] == nullNode ) {
570         return 0;
571       }
572       else mySize++;
573     }
574     else {
575       ///jsw_rbnode_t head = {0}; /* False tree root */
576       //RedBlackNode<Comparable> head = new RedBlackNode<Comparable>;
577       RedBlackNode<Comparable> head; /* False tree root */
578       //RedBlackNode<Comparable> *head; /* False tree root */
579       RedBlackNode<Comparable> *g, *t;      /* Grandparent & parent */
580       RedBlackNode<Comparable> *p, *q;      /* Iterator & parent */
581       int dir = 0, last = 0;
582
583       /* Set up our helpers */
584       t = &head;
```

```
585       ///cout << "hi5" << endl;
586       g = p = nullNode;
587       q = t->link[1] = header->link[1];
588       ///cout << "hi6" << endl;
589
590       /* Search down the tree for a place to insert */
591       for ( ; ; ) {
592         if ( q == nullNode ) {
593         /* Insert a new node at the first null link */
594         p->link[dir] = q = new RedBlackNode<Comparable>( x, nullNode, nullNode, bx, by, x0x, x0y,
                x1x, x1y );
595
596         if ( q == nullNode )
597           return 0;
598         else mySize++;
599         }
600         else if ( is_red ( q->link[0] ) && is_red ( q->link[1] ) ) {
601         /* Simple red violation: color flip */
602         q->red = 1;
603         q->link[0]->red = 0;
604         q->link[1]->red = 0;
605         }
606
607         if ( is_red ( q ) && is_red ( p ) ) {
608         /* Hard red violation: rotations necessary */
609         int dir2 = t->link[1] == g;
610
611         if ( q == p->link[last] )
612           t->link[dir2] = jsw_single ( g, !last );
613         else
614           t->link[dir2] = jsw_double ( g, !last );
615         }
616
617         /*
618         Stop working if we inserted a node. This
619         check also disallows duplicates in the tree
620         */
621         if ( q->element == x )
622         break;
623
624         last = dir;
625         dir = q->element < x;
626
627         /* Move the helpers down */
628         if ( g != nullNode )
629         t = g;
630
631         g = p, p = q;
632         q = q->link[dir];
633       }
634
635       /* Update the root (it may be different) */
636       header->link[1] = head.link[1];
637     }
638
639     /* Make the root black for simplified logic */
640     header->link[1]->red = 0;
641
642     return 1;
643   }
644
645   ///**
646   // * Insert item x into the tree. Does nothing if x already present.
647   // */
648   //template <class Comparable>
649   //void RedBlackTree<Comparable>::insert( const Comparable & x, const double & bx,
650   //    const double & by, const double & x0x, const double & x0y, const double & x1x,
651   //    const double & x1y )
652   //{
653   //  current = parent = grand = header;
```

```
654  //   nullNode->element = x;
655  //
656  //   while( current->element != x )
657  //   {
658  //     great = grand; grand = parent; parent = current;
659  //     current = x < current->element ?  current->left : current->right;
660  //
661  //       // Check if two red children; fix if so
662  //     if( current->left->color == RED && current->right->color == RED )
663  //        handleReorient( x );
664  //   }
665  //
666  //     // Insertion fails if already present
667  //   if( current != nullNode )
668  //     return;
669  //   current = new RedBlackNode<Comparable>( x, nullNode, nullNode, bx, by, x0x, x0y, x1x, x1y );
670  //
671  //     // Attach to parent
672  //   if( x < parent->element )
673  //     parent->left = current;
674  //   else
675  //     parent->right = current;
676  //   handleReorient( x );
677  //}
678
679
680  ///**
681  // * Internal routine that is called during an insertion
682  // *      if a node has two red children. Performs flip
683  // *      and rotatons.
684  // * item is the item being inserted.
685  // */
686  //template <class Comparable>
687  //void RedBlackTree<Comparable>::handleReorient( const Comparable & item )
688  //{
689  //     // Do the color flip
690  //   current->color = RED;
691  //   current->left->color = BLACK;
692  //   current->right->color = BLACK;
693  //
694  //   if( parent->color == RED )   // Have to rotate
695  //   {
696  //     grand->color = RED;
697  //     if( item < grand->element != item < parent->element )
698  //       parent = rotate( item, grand );  // Start dbl rotate
699  //     current = rotate( item, great );
700  //     current->color = BLACK;
701  //   }
702  //   header->right->color = BLACK; // Make root black
703  //}
704  //
705  ///**
706  // * Internal routine that performs a single or double rotation.
707  // * Because the result is attached to the parent, there are four cases.
708  // * Called by handleReorient.
709  // * item is the item in handleReorient.
710  // * parent is the parent of the root of the rotated subtree.
711  // * Return the root of the rotated subtree.
712  // */
713  //template <class Comparable>
714  //RedBlackNode<Comparable> *
715  //RedBlackTree<Comparable>::rotate( const Comparable & item,
716  //              RedBlackNode<Comparable> *theParent ) const
717  //{
718  //   if( item < theParent->element )
719  //   {
720  //     item < theParent->left->element ?
721  //       rotateWithLeftChild( theParent->left )  :  // LL
722  //       rotateWithRightChild( theParent->left ) ;  // LR
723  //     return theParent->left;
```

```
724   //  }
725   //  else
726   //  {
727   //    item < theParent->right->element ?
728   //      rotateWithLeftChild( theParent->right ) :  // RL
729   //      rotateWithRightChild( theParent->right );  // RR
730   //    return theParent->right;
731   //  }
732   //}
733   //
734   ///**
735   // * Rotate binary tree node with left child.
736   // */
737   //template <class Comparable>
738   //void RedBlackTree<Comparable>::
739   //rotateWithLeftChild( RedBlackNode<Comparable> * & k2 ) const
740   //{
741   //  RedBlackNode<Comparable> *k1 = k2->left;
742   //  k2->left = k1->right;
743   //  k1->right = k2;
744   //  k2 = k1;
745   //}
746   //
747   ///**
748   // * Rotate binary tree node with right child.
749   // */
750   //template <class Comparable>
751   //void RedBlackTree<Comparable>::
752   //rotateWithRightChild( RedBlackNode<Comparable> * & k1 ) const
753   //{
754   //  RedBlackNode<Comparable> *k2 = k1->right;
755   //  k1->right = k2->left;
756   //  k2->left = k1;
757   //  k1 = k2;
758   //}
759
760
761   /**
762    * Internal method to reclaim internal nodes
763    * in subtree t.
764    */
765   template <class Comparable>
766   void RedBlackTree<Comparable>::reclaimMemory( RedBlackNode<Comparable> *t ) const
767   {
768     if( t != t->link[0] )
769     {
770       reclaimMemory( t->link[0] );
771       reclaimMemory( t->link[1] );
772       delete t;
773     }
774   }
775
776
777   /**
778     <summary>
779     Checks the color of a red black node
780     <summary>
781     <param name="root">The node to check</param>
782     <returns>1 for a red node, 0 for a black node</returns>
783     <remarks>For jsw_rbtree.c internal use only</remarks>
784   */
785   template <class Comparable>
786   int RedBlackTree<Comparable>::is_red ( RedBlackNode<Comparable> *root ) const
787   {
788     return root != nullNode && root->red == 1;
789   }
```

References

[1] S. Munier, Phys. Rept. 473, 1 (2009).

[2] K. Itakura, Nucl. Phys. A 774, 277 (2006).

[3] G. Chew, S. Frautschi, Phys. Rev. Lett. 7, 394 (1961).

[4] G. Chew, S. Frautschi, Phys. Rev. Lett. 8, 41 (1962).

[5] A. Donnachie, P. Landshoff, Phys. Lett. B 595, 393-399 (2004).

[6] J. R. Forshaw, D. A. Ross, *Quantum Chromodynamics and the Pomeron*. Cambridge Lecture Notes in Physics 9, (1997).

[7] S. Donnachie, G. Dosch, P. Landshoff, O. Nachtmann, *Pomeron Physics and QCD*. Cambridge University Press, (2002).

[8] R. J. Eden, P. V. Landshoff, D. I. Olive, J. C. Polkinghorne, *The Analytic S-Matrix*. Cambridge University Press, (1966).

[9] I. Gradshteyn, I. Ryzhik, *Table of Integrals, Series, and Products, 6E*. Academic Press, (2000).

[10] G. Soyez, *Deep Inelastic Scattering at small x*. Dissertation, Université de Liege, (2004).

[11] A. H. Mueller, Nucl. Phys. B 415, 373 (1994).

[12] A. H. Mueller, Nucl. Phys. B 437, 107 (1995).

[13] A. H. Mueller, B. Patel, Nucl. Phys. B425, 471-488 (1994).

[14] S. Munier, R. Peschanski, Phys. Rev. D 69, 034008 (2003).

[15] S. Munier, R. Peschanski, Phys. Rev. Lett. 91, 232001 (2003).

[16] S. Munier, R. Peschanski, Phys. Rev. D70, 077503 (2004).

[17] A. H. Mueller, A. I. Shoshi, Nucl. Phys. B 692, 175 (2004).

[18] E. Brunet, B. Derrida, Phys. Rev. E 57, 2597 (1997).

[19] E. Brunet, B. Derrida, A. H. Mueller, S. Munier, Phys. Rev. E 73, 056126 (2006).

[20] M. Ciafaloni, D. Colferai, G. P. Salam, JHEP 9910, 017 (1999).

[21] A. Mueller, G. P. Salam, Nucl. Phys. B 475, 293 (1996).

[22] G. P. Salam, Nucl. Phys. B 461, 512 (1996).

[23] G. P. Salam, Comput.Phys.Commun. 105, 62 (1997).

[24] S. Munier, G. P. Salam, G. Soyez, Phys. Rev. D 78, 054009 (2008).

[25] A. H. Mueller, S. Munier, Phys. Rev. D 81, 105014 (2010).

[26] S. Munier, Proceedings of DIS 2010, Florence, Italy, (2010).

[27] C. Marquet, H. Weigert, Nucl. Phys. A 843, 68 (2010).

[28] E. Iancu, A. H. Mueller, S. Munier, Phys. Lett. B 606, 342 (2005).

[29] E. Iancu, D. N. Triantafyllopoulos, Nucl. Phys. A 756, 419 (2005)

[30] E. Iancu, D. N. Triantafyllopoulos, Phys. Lett. B 610, 253 (2005).

[31] C. Ewerz, O. Nachtmann, Annals Phys. 322, 1635-1669 (2007).

[32] C. Ewerz, O. Nachtmann, Annals Phys. 322, 1670-1726 (2007).

[33] N. Nikolaev, B. G. Zakharov, Z. Phys. C49, 607-618 (1991).

[34] N. Nikolaev, B. G. Zakharov, Z. Phys. C53, 331-346 (1992).

[35] K. Golec-Biernat, M. Wüsthoff, Phys. Rev. D 59, 014017 (1999).

[36] K. Golec-Biernat, M. Wüsthoff, Phys. Rev. D 60, 114023 (1999).

[37] K. Golec-Biernat, M. Wüsthoff, Eur. Phys. J. C 20, 313 (2001).

[38] C. Marquet, L. Schoeffel, Phys. Lett. B639, 471-477 (2006).

[39] Y. Kovchegov, Phys. Rev. D60, 034008 (1999).

[40] Y. Kovchegov, Phys. Rev. D61 074018 (2000).

[41] I. Balitsky, Nucl. Phys. B463 99-160 (1996).

[42] E. Levin, J. Bartels, Nucl. Phys. B387, 617 (1992).

[43] E. Iancu, K. Itakura, L. McLerran, Nucl. Phys. A 708, 327 (2002).

[44] E. Levin, K. Tuchin, Nucl. Phys. A 691, 779-790 (2001).

[45] W. Saarloos, Phys. Rept. 386, 29-222 (2003).

[46] M. Bramson, Memoirs of the American Mathematical Society 285, (1983).

[47] K. Golec-Biernat, L. Motyka, A. Stasto, Phys. Rev. D65, 074037 (2002).

[48] L. Gribov, E. Levin, M. Ryskin, Nucl. Phys. B188, 1-150 (1983).

[49] G. T'Hooft, Nucl. Phys. B72, 461-473 (1974).

[50] R. Sedgewick, *Algorithms in C++, 3E.* Addison-Wesley Publishing Company, Inc., (1998).

[51] J. Walker, *Red Black Tree Tutorial.* <http://eternallyconfuzzled.com/tuts/datastructures/jsw_tut

[52] *Lonestar User Guide.* <http://www.tacc.utexas.edu>.

[53] H. Politzer, Phys. Rep. 14, 129-180 (1974).

[54] D. Gross, F. Wilczek, Phys. Rev. D8, 3633-3652 (1973).

[55] L. Lukaszuk, A. Martin, Il Nuovo Cimento 52, 122 (1967).

[56] J. Jalilian-Marian, A. Kovner, A. Leonidov, H. Weigert, Nucl. Phys. B504, 415-431 (1997).

[57] E. Iancu, A. Leonidov, L. D. McLerran, Phys. Lett. B510, 133-144 (2001).